7 Architecture, patterns and components

with Alan O'Callaghan and Alan Cameron Wills

A spider conducts operations that resemble those of a weaver, and a bee puts to shame many an architect in the construction of her cells. But what distinguishes the worst architect from the best of bees is this, that the architect raises his structure in imagination before he erects it in reality.

K. Marx (*Capital, Volume I*)

This chapter examines various issues that arise in modern object-oriented development. In particular, we examine the emerging discipline of software architecture and review current debates from the particular perspective of the object-oriented designer who, arguably, has more choice open to her than most other software developers. In the context of that discussion we then look at the phenomenon of patterns, which are regarded as essential for good object-oriented design. Lastly we look at the application of object-oriented ideas to component based development, which is the area where we expect to see the delivery of most of the benefits discussed in Chapter 2. As in Chapter 6, our treatment of component based development follows the approach of Catalysis closely.

7.1 Software and system architecture

Software architecture is a hot topic in software development generally as well as within object-orientation. It has been palpably growing in importance, for reasons we discuss below, since at least 1992 with a clear separation in schools of thought surrounding its nature appearing more recently. However the metaphor has a much longer genealogy than the modern debate is sometimes prepared to admit. The notion of software architecture has a pedigree at least as long as those of 'software engineering' and the 'software crisis'. These latter two terms are commonly accepted to have originated in the famous NATO conference of 1968 (Naur, 1969). Fred Brooks Jr. refers in a number of the influential articles in his Mythical Man-Month collection to architecture (Brooks, 1975), but credits Blaauw (Blaauw, 1970) with the first use of the term five years previously. However, according to Ian Hugo, a delegate at that historic NATO conference, the idea of 'software architecture' and the rôle of 'software architects' were common currency in its deliberations (Hugo, personal communication with Alan O'Callaghan) – the analogy was, however, considered by the proceedings' editors too fanciful a notion to be reflected in the official minutes. It is symbolic indeed that the idea co-originates with that of the software crisis and that it was largely discarded in the intervening period, while the discipline of software engineering as it has been practised for more than three decades has manifestly failed to resolve the situation it was ushered in to redress. It is perhaps unsurprising then that the modern discussion of software architecture lies at the heart of a debate which is reassessing the very basis of the discipline(s) of software and system development. And in that debate, experience of both object-orientation and software patterns provides critical insights.

ARCHITECTURE AS GROSS STRUCTURE

There is as yet no clear and unambiguous definition of what software architecture is. What consensus does exist to date seems to centre around issues of high-level design and the gross structure of systems, including both their static and dynamic properties. Larry Bass and his colleagues (Bass, *et al.*, 1998), for example, in a since oft-quoted definition, say that:

> 'The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them.'

Mary Shaw and David Garlan in their influential short volume on software architecture go so far as to date its origins from the moment software systems were first broken into modules (Shaw and Garlan, 1996). At the same time they recognize that, historically, such architectures have been implicit. In attempting to render architecture more explicit and formal, they introduce the important notions of software *architectural style* and of *Architectural Description Languages* (ADLs).

They treat the architecture of a specific system as a collection of computational components together with a description of the interactions, or connectors, linking these components. An architectural style, according to Shaw and Garlan, defines a family of systems in terms of this structural organization. Clients, servers, filters, layers and databases are all given as possible examples of components, while example connectors are procedure calls, event broadcasts, database protocols and pipes. Table 7.1, which is adapted from their book, gives a list of common architectural styles.

Shaw and Garlan's position and that of others at Carnegie Mellon University

(CMU) and at the Software Engineering Institute (SEI) which it is home to such as Bass *et al.* , may be characterized as the view that architecture is equivalent to the high level structure of a system. The inclusion of OO in the table itself highlights an issue with an approach that reduces architecture to one of *components plus connectors*. Many of us believe that architecture is more than just boxes and lines, however much semantics is attached to them. To make sense it must include a notion of architectural vision: the unifying concept that we all recognize equally when we see an elegant software system or a Nash terrace or, more importantly, when we have to build one.

Table 7.1 Common architectural styles (after Shaw and Garlan)

Dataflow Systems	***Virtual Machines***
Batch sequential	Interpreters
Pipes and filters	Rule-based systems
Call-and-Return systems	***Data-centred systems (repositories)***
Main program and subroutine	Databases
OO systems	Hypertext systems
Hierarchical layers	Blackboards
Independent Components	
Communicating processes	
Event systems	

Shaw and Garlan focus on the implementation technology and regard objects merely as instances of abstract data types. They note that an object is both responsible for preserving its representation and simultaneously hiding that representation from its clients. However, they seem to miss the significance of this; for, as Cook (1994...) and O'Callaghan (1994...) amongst others have pointed out, the fact that an object presents itself as abstract behaviour (through its operations) encapsulating both the implementation of its operations and the (localized) data they manipulate isolates the programmer from the digital architecture of the underlying machine. As a consequence of this freedom from the constraints of Von Neumann architecture developers can use objects as building blocks to create any number of software architectures, especially ones which reflect the vocabulary of the problem space. Cook says that objects are in this sense 'architecture free'. This means, amongst other things, that objects can be used to build pipes-and-filter structures, layered architectures, blackboard systems and, arguably, any of the other styles listed by Shaw and Garlan.

The reduction of object-orientation to just one of a number of implementation styles further ignores Graham's (1998) identification of objects as a 'general knowledge acquisition mechanism'. This means that the use of objects in spaces other than the implementation space is ignored, and therefore their potential significance for software architecture grossly underemphasized. This seems to be an important gap in the current orthodoxy of software architecture thinking since often, it is precisely this 'modellability' of objects and the promise of systems that are consequently flexible to business change that makes them an attractive proposition. When Shaw and Garlan contrast what they call 'implicit invocation' architecture (i.e. actor or blackboard systems where objects register interest in events) with object technology, they seem to be unwilling to admit that implicit invocation is readily modelled using object-oriented methods. Their view of object-orientation seems to be limited to the models of current object-oriented programming languages. Yet in the same work they use object-oriented design for their architectural design tool AESOP – implicitly admitting that OO can describe other styles. Worse still, they criticize OO for not being able to represent stylistic constraints. This narrow view of OO excludes object-oriented methods where objects with embedded rulesets can represent such things as semantic integrity constraints and the like.

On the other hand, Shaw and Garlan do present some sound arguments for the extension of the object-oriented metaphor. It is argued, for example, that ADLs imply a need for rôle modelling (and thus dynamic classification). Of course this does not mean that objects are the best choice for every requirement that can be imagined. Through the use of case studies such as Tektronix Inc.'s development of a reusable architecture for oscilloscopes, and another on mobile robotics, Shaw and Garlan effectively demonstrate that different topologies reflect different costs and benefits and therefore have different applicability. But they are somewhat unconvincing in arguing that components and connectors add up to architecture. For the practising object technologist such a reductionist view of architecture is of little help in helping her create software systems when she has so many different architectural possibilities open to her.

The other significant contribution made in the Shaw and Garlan book is to introduce ADLs. The authors point out that structural decomposition is traditionally expressed either through the modularization facilities of programming languages or through the special medium of Module Interconnexion Languages (MILs). These are criticised for being too low-level in their descriptions of interconnexions between different computational elements and because 'they fail to provide a clean separation of concerns between architectural-level issues and those related to the choices of algorithms and data structures' (page 148). Newer, component-based languages such as Occam II (Pountain, 1989) and Connection (Mak, 1992) or environments such as STEP (Rosene, 1985) which enforced specialized structural patterns of organization are criticised for their narrow scope. Instead six key requirements for a higher-level kind of language, the ADL, are enumerated as follows.

1. *Composition*. It should be possible to describe a system as a composition of independent components and connectors

2. *Abstraction*. It should be possible to describe the components and their interactions within software architecture in a way that clearly and explicitly prescribes their abstract roles in a system.
3. *Reusability*. It should be possible to reuse components, connectors, and architectural patterns in different architectural descriptions, even if they were developed outside the context of the architectural system.
4. *Configuration*. Architectural descriptions should localize the description of system structure, independently of the elements being structured. They should also support dynamic reconfiguration.
5. *Heterogeneity*. It should be possible to combine multiple, heterogeneous architectural descriptions.
6. *Analysis*. It should be possible to perform rich and varied analyses of architectural descriptions.

Shaw and Garlan point out, quite correctly, that the typical box-and-line diagrams that often pass for 'architectural description' focus on components often to the virtual, and sometimes actual, exclusion of the connectors between them. They are therefore underspecified in a number of crucial contexts, notably third-party 'packaged' components, multi-language systems, legacy systems and, perhaps most critically of all, large-scale real-time embedded systems. Additionally the boxes, lines and the adjacency between boxes lack semantic consistency between diagrams and sometimes even within the same diagram and ignore the need to structure interface definitions. This points to a minimum of two levels of structure and abstraction that are typically missing: abstractions for connexions, and segmentation of interfaces. As it stands, such diagrams rely heavily on the knowledge and experience of the person in the architect's role, and this is held informally.

UniCon (short for 'Universal Connector language') is an example ADL developed at Carnegie Mellon University (Shaw *et al*., 1995). It is characterized amongst other things by having an explicit connector construct which represents the rules by which components can be hooked up, and by having a defined abstraction function which allows an architectural description to be retrieved from low-level constructs. A component has players – the entities visible and named in its interface; a connector has roles – the named entities in its protocol- that have to be satisfied by players. UniCon supports the checking of associations between players and roles, the checking of the types of components and connectors themselves, and the adherence of the components and connectors to architectural styles.

UniCon and other ADLs have undoubtedly added rigour and formality to the description and analysis of structure. This is of first rate importance in certain classes of application, notably real-time systems. In this context, Selic *et al*. (1994) argue for the importance of an approach to architectural design in their real-time ROOM method and the ObjecTime tool that supports it. ObjecTime and Rational Software Inc. have been working together since 1997 to align their respective technologies and, following a period of strategic co-operation in which ObjecTime were Rational's exclusive supplier of modelling tools and automatic code generation technology for the real-time domain (and Rational had world-wide distribution rights for ObjectTime Developer) Rational Software announced the acquisition of ObjecTime in December 1999. As a result the architectural description concepts that originated in ROOM have now found their way into real-time extensions for UML (Selic and Rumbaugh, 1998). Arguably, UML with real-time extensions is the most widely used commercial-strength ADL. UML models the structure of a system by identifying the entities of interest and the relationships between them. In the category of real-time systems that form the domain described by ROOM (complex, event-driven and potentially distributed systems such as those that occur in telecommunications, aerospace and defence systems) two of UML's total of nine diagram types are the focus of the real-time extensions: class diagrams and collaboration diagrams. Class diagrams in UML capture relationships that are universal, that is they apply to all possible instances in all possible contexts. Collaboration diagrams on the other hand capture relationships in just one particular context. Collaboration diagrams therefore are able to distinguish between different usages of different instances of the same class. This notion is captured in the concept of a *rôle*. Typically the complete specification of a complex real-time system in terms of its structure is obtained, using these extensions, through a combination of class diagrams and collaboration diagrams.

Three constructs are defined in order to use UML as an ADL for such systems:

- Capsules (formerly known as *actors* in ROOM – and not to be confused with the concept of the same name used with use cases);
- Ports;
- Connectors (formerly known as *bindings* in ROOM).

A **capsule** is a complex, physical, possibly distributed architectural object that interacts with its environment through one or more signal-based boundary objects called ports. It completely contains all its ports and sub-capsules. That is, these cannot exist independently of their 'owning' capsule (unless the sub-capsule is a 'plug-in' capsule). This makes the capsule the central modelling element in real-time extended UML. A **port** is also a physical object, part of a capsule that implements a specific interface and plays a specific role in some collaboration between capsules. Being physical objects, ports are visible from the inside and from the outside of capsules. From the outside ports are distinguishable from each other only by their identity and the role that they play in their protocol. From the inside, however, ports can either be *relay* ports or *end* ports. Relay ports differ from end ports in their internal connexions. Relay ports are connected, through other ports, to subcapsules and exist to simply pass signals through. End ports, in contrast, are connected to a capsule's state machine and are the ultimate sources and sinks of signals. Both capsules and ports are modelled with stereotyped class icons in UML, but an additional stereotype icon, a small black square (or, alternatively, a white one for a conjugated port in a binary protocol) is available. A *connector* is a physical communication channel that provides the transmission facilities for a particular, abstract, signal-based protocol. A connector can only interconnect ports that play complimentary roles in the protocol with which it is associated. Connectors are modelled with UML associations. Figure 7.1 shows the visual syntax of these constructs in UML extended for real-time. The approach forces capsules to communicate solely through their ports, permitting the decoupling of the capsules' internal representations from any knowledge about their surrounding environment, and enabling reuse.

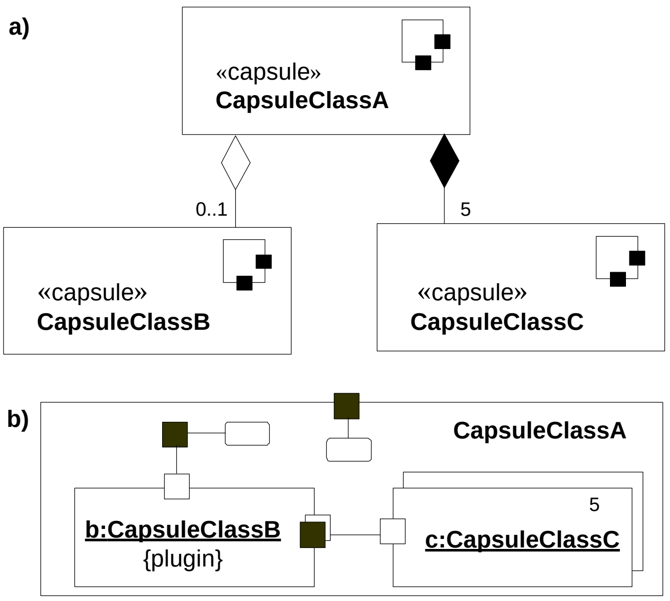


Figure 7.1 UML for Real-Time as an ADL (adapted from Rumbaugh and Selic, 1998): **(a)** A UML-RT class diagram; **(b)** one possible corresponding collaboration diagram.

The broader question arises, however, as to how much increased formality in and of itself will help in addressing the fundamental problems of software development. The roots of the research into architecture at Carnegie Mellon lie in Mary Shaw's pursuit of a definition of the Software Engineering discipline itself (e.g. Shaw, 1990). She opines that the maturity of an engineering discipline is marked by the emergence of a 'sufficient scientific basis' to enable a critical mass of science-trained professionals to apply their theory to both the analysis of problems and the synthesis of solutions. Further progress is observed when science becomes a forcing function. She presents, therefore, a model that contends that the emergence of a sound engineering discipline of software depends on the vigorous pursuit of applicable science and then the reduction of that science to a practice.

This linear model of science preceding practice and reducing engineering to 'mere' problem-solving is, of course, the received wisdom of traditional Computer

Science. But the problem is what is the ‘applicable science’? There is a growing challenge to the old orthodoxy which sees computing as essentially a specialized branch of mathematics. Borenstein (1991), for example, in his marvellously named book *Programming as if People Mattered* notes that the most interesting part of the process of building software is the human part, its design and its use: ‘Inasmuch as human-oriented software engineering is the study of this process, it could be argued that it is more properly a branch of anthropology than of mathematical sciences. The study of software creation may, in fact, be grossly mis-classified in the academic world today, leading to a distorted overemphasis on formal models and a lack of basic work in collecting raw material that comes, most often, in anecdotal (or at least non-quantitative) form.’ (p.36)

Interestingly Borenstein upholds what Shaw and Garlan, and indeed everyone else at CMU and the SEI appears to bemoan, the informality and unquantifiable characteristics of current software design practice. Bruce I. Blum, a former VP of Wolf Research and an erstwhile research scientist at John Hopkins University has called for the redesign of the disciplines of Computer Science and Software Engineering on similar grounds to those advocated by Borenstein (Blum, 1996; 1998). In his work he has questioned the epistemology of the received orthodoxy on both the grounds of the philosophy of science and of practice. He points to a dialectical contradiction in the making of software that traditional Computer Science is blind to. He distinguishes between the ‘program-in-the-computer’ and the ‘program-in-the-world’. The program in the computer is subject to the closed abstractions of mathematics and is, theoretically at least, verifiable against some specification. Indeed the core of interest in the ‘program-in-the-computer’ is the extent of the difficulty of its construction. But what is in the computer must also exist, on completion, as a ‘program-in-the-world’ where the one and only test is its usefulness. Its value depends upon its ability to transform the pre-existing situation into which it now inserts itself, both impacting upon and being impacted by that situation. As a result the ‘program-in-the-computer’ exists in a stable, well-known and formally representable environment, but the very same software as ‘the program-in-the-world’ is dynamic and incompletely understood. A key issue is this: formal approaches work primarily by a process of selection from an existing or theoretical universe of extant, well-defined formal abstractions. Selection is easier than creating from scratch and therefore all such gains are valuable, but a total solution using formal methods is possible only once *all possible* design solutions have been formalized. Hence, no doubt, the SEI’s interest in collecting architectural styles. However, Blum’s criticism suggests that such a task is not only an Herculean labour but that, because it too rests on a mythology – the received wisdom of Computer Science – it can never be completed.

PROBLEM FRAMES

A more profound approach is due to Jackson (1995). He defines a problem frame as a structure consisting of principal parts and a solution task. The principal parts correspond to what I have called agents, business objects, conversations, actions and use cases. The solution task is the work one has to do to meet some requirement concerning these parts or objects. He then abstracts from the objects to problem domains and the phenomena that are shared between them: in the sense of the elements that can be described in the languages of both domains. More importantly, each frame may have a set of rules that connect pairs of domains. I think that problem frames are a not only a requirements engineering technique but potentially also an architectural technique because they describe not a solution but a suitable approach to finding a solution. They also suggest the patterns that we deal with in the next section. The idea is focused more on the problems of the requirements analyst trying to understand a problem and select an approach than the object-oriented designer who has already selected the architectural style and implementation technology.

Typical problem frames include the following:

- CONNEXION: introduces a separate problem domain between the application and solution domains. Examples: a post office; CORBA.
- JSP: helps to describe a program in terms of its input and output streams. Example: a typical accounting or stock control system.
- SIMPLE CONTROL: describes the situation where known control rules are applied to a controllable phenomenon. Examples: embedded real-time controllers; vending machines.
- SIMPLE INFORMATION SYSTEMS: the real-world analogue of the JSP frame, the problem concerns users requesting and updating information about some domain. Example: database systems.
- WORKPIECES: describes the way operators' commands are linked to the manipulation of objects. Example: text editors.

Problem frames taken together represent a pattern language (see below) since realistic problems usually involve several frames. Jackson argues that identifying the frame is a precursor to selecting an appropriate method. He characterizes, by way of example, the use case approach as such a method and points to some of its limitations – as we shall in Chapter 8 – with the aim of showing that its use is restricted to problem frames where user I/O dominates. This is quite correct, but we feel that viewing object modelling as a form of knowledge representation frees me from this criticism. Although object technology undoubtedly has frame dependent limitations, it can be specialized to deal with quite a variety of frames. The main reason for this is the semantic richness provided by rulesets and, for example, the inclusion of invariance conditions as well as pre- and post-conditions.

We would encourage readers to think about their own set of familiar problems and ask if there are other frames at this level of generality. Our contribution and experience are beginning to suggest that there is a frame that abstracts our recurring order processing or trading problem, one for modelling agents (called Assistant) and one for general simulation problems. Describing these frames in Jackson‘s language of domains and shared phenomena is left as an exercise for the interested reader

Picking up similar themes from different roots, those of Syntropy (Cook and Daniels, 1994), O’Callaghan (2000a) has argued that object technology, to deliver its full benefits, needs to be used for modelling in two conceptually distinct spaces: the problem space, which is a conceptual model of the real world situation, and the solution space where the software part of the solution is created. Object types capture observed behaviour of key abstractions in the problem space, but specify the behaviour of a to-be-designed solution in the implementation space. The programmer controls the latter, but can never do so in the former. Despite the ‘seamlessness’ of the use of the object as the basic metaphor in both spaces, the fundamental differences in the nature of the spaces themselves and therefore of the abstractions that are modelled within them, makes traceability a non-trivial issue. He posits the need for a Janus-like entity which sits between these two spaces, looking both ways at once. It is this he calls the software architecture. What is interesting is that while Borenstein, Blum and O’Callaghan, to which we might add the voices of Mitch Kapor (Kapor, 1990) and Terry Winograd (Winograd, 1996), appear to be coming from left field (what the fuck is left field…) with respect to the currently dominant view of software architecture, they actually have more in common with the historical roots of the notion of software architecture than does the SEI.

Fred Brooks Jr., in his article *Aristocracy, Democracy and System Design* (in Brooks, 1975) argued that the single most important consideration in system design is its conceptual integrity. This was for him the defining property of software or system architecture, and the chief charge of the architect. Brooks stated that it is far better to reflect a coherent set of design ideas, and if necessary omit feature or functionality in order to maintain that coherence, than it is to build a system that contains heterogeneous, or uncoordinated concepts, however good they each may be independently. This raises the following issues.

- How is conceptual integrity to be achieved?
- Is it possible to maintain conceptual integrity without separation between a small, ‘architectural’ elite and a larger group of plebeian implementers for a system?
- How do you ensure the architects do not deliver unimplementable or over costly implementations?
- How do you ensure that the architecture is reflected in the detailed design?

It was in answering these questions that Brooks made a conscious appeal to the metaphor of architecture in the built environment. He carefully distinguished between architecture and implementation. Echoing Blaauw he gave the simple example of a clock, whose architecture consists of its face, its hands and its winding knob. Having learned the architecture of a clock a child can tell the time whether using a wristwatch, a kitchen clock or a clock tower. The mechanisms by which the time-telling function is delivered is a detail of implementation and realization, not architecture.

‘By the architecture of a system’ says Brooks, ‘I mean the complete and detailed specification of the user interface. For a computer this is the programming manual. For a compiler it is the language manual. For a control program it is the manuals for the language or languages used to invoke its functions. For the entire system it is the union of the manuals the user must consult to do his entire job’ (p.45).

While this definition seems inadequate today, it has at least the virtue of establishing that the architect is the client’s agent not the developers’. It strongly suggests that the conceptual integrity of the system, represented by the architecture, gains its shape from the perceived needs of the client and that the final test of what constitutes a ‘good’ architecture is its usefulness. In this important aspect there is a strong thread of continuity between the ideas of Brooks and Blum, although ironically Blum does not use the term ‘software architecture’, and a symmetrical discontinuity between Brooks and the SEI despite their claim to it. It is also worth noting that this view of the software architect as the client’s agent is also a preoccupation of the recently founded World Wide Institute of Software Architects (WWISA, 2000).

Again following Blaauw, Brooks suggests that the overall creative effort involves three distinct phases: architecture, implementation and realization. For Brooks, architecture appears to end with the complete and final specification of the system; the design ‘of module boundaries, table structures, pass or phase breakdowns, algorithms, and all kinds of tools’ belongs to implementation. Brooks believed that all three phases could occur to some extent in parallel and that successful design requires an ongoing conversation between architects and implementers. Nevertheless the boundaries of the architects’ input to the dialogue are confined to the external specification of the system. There is clearly some tension between this idea of architect’s role and Brooks’ insistence on the need to maintain the conceptual integrity of the system. In modern systems at least, which involve issues of scale and distribution which did not exist in 1975, traceability of a construction back through its structure to its specification and to the need which spawned it is crucial in maintaining the integrity of the system.

ARCHITECTURE AS DESIGN RATIONALE

There appears to be the need for a marriage between the idea of the architect as the client’s agent championing the conceptual integrity of the system on the one hand, and ideas about the internal structures of the system on the other hand. Many modern theorists of Software Architecture draw inspiration from a seminal paper written by Dewayne Perry and Alexander Wolf (Perry and Wolf, 1992). They defined software architecture in terms of an equation:

Software Architecture = {Elements, Form, Rationale}.

Barry Boehm has apparently qualified the last of these three terms to read ‘Rationale/Constraints’. A powerfully influential interpretation of this idea, applied specifically to object-oriented development, has been offered by Phillipe Kruchten in his ‘4+1 View Model’ of software architecture (Kruchten, 1995). This view underpins the Unified Process and is responsible, in large part, for the claims it makes to being ‘architecture-centric’. Kruchten concedes that software architecture deals with abstraction, with composition and decomposition, and also with style and aesthetics. To deal with all these aspects, especially in the face of large and challenging systems’ developments, Kruchten proposes a generic model made up of five different views:

- The *logical* view: an object model of the design
- The *process* view: which models the concurrency and synchronization issues of the design
- The *physical* view: a model of the mapping of software onto the hardware elements of the solution, including distribution issues
- The *development* view: the static organization of the software in its development environment
- A *scenarios*-based view: the usage scenarios from which the architecture is partially derived, and against which it is validated.

Kruchten applies the Perry and Wolf equation independently on each view. The set of elements in terms of components, containers and connectors are defined for each view, as are the forms and patterns which work in the particular context in which the architecture is to be applied. Similarly, the rationale and constraints for each view are also captured, connecting the architecture to the requirements. Each view is captured in the form of a blueprint in a notation appropriate to the view (in the original paper, which predated the UML, subsets of the Booch notation were used for each view), and may have an architectural style, *a la* Shaw and Garlan, to it.

Criticizing a rather linear, four-phase (sketching, organizing, specifying and optimising), twelve-step process for architecture proposed by Witt *et al.* (1994), Kruchten proposes an iterative, scenario-driven approach instead. Based on relative risk and criticality, a small number of scenarios are chosen for an iteration. Then a straw man architecture is put in place. Scenarios are scripted to derive the major abstractions (classes, collaborations, processes, subsystems etc.) and then decomposed into object-operation pairs. The architectural elements that have been discovered are laid out on to the four blueprints: logical, physical, process and development. The architecture is then implemented, tested, measured and analysed, possibly revealing flaws or opportunities for enhancement. Subsequent iterations can now begin. The documentation resulting from this process is in fact two sets of documents: A *Software Architecture Document* (the recommended template for which is shown in Figure 7.2) and a separate *Software Design Guidelines* which captures the most important design decisions that must be respected to maintain the conceptual integrity of the architecture. The essence of the process is this: the initial architectural prototype evolves to be the final system.

Astute readers will recognise in this approach the bones of the Unified Process (Jacobson 1999), itself a public domain subset of the proprietary Rational Unified

Process, as well as a submission to the OMG's ongoing effort to define a standard, generic process for object-oriented development. In its modern, post-UML form Kruchten's '4+1' model has been retained but with some renaming. The logical, process views and development views remain, but the physical view is now called the Implementation View and/or the Component View (Grady Booch has recently used these names interchangeably in the same presentation) and the scenarios-based view is called the Use Case View, firmly tying the '4+1 Model' into the UML as well as the Unified Process. In a keynote speech to UML World, Grady Booch has gone further and defined an architecture metamodel which is depicted in Figure 7.3 (Booch 1998??….). We will return to it later, but here we can note that the metamodel states that a *Software Architecture* is represented by a *Software Architecture Description* composed of *Architectural Views* (*Logical*, *Process*, *Implementation*, *Deployment* and *Use Case*) each depicted in an *Architectural Blueprint* and an *Architectural Style Guide*. It also shows a binary relationship between *Software Architecture* and *Requirements.* Allowing for a renaming of Kruchten's original *Software Design Guide* as *Architectural Style Guide* the metamodel is entirely consistent with the '4+1 Model'.

Title Page
Change History
Table of Contents
List of Figures
● 1. Scope
● 2. References
● 3. Software Architecture
● 4. Architecture Goal and Constraints
● 5. Logical Architecture
● 6. Process Architecture
● 7. Development Architecture
● 8. Physical Architecture
● 9. Scenarios
● 10. Size and Performance
● 11. Quality
Appendices
❍ A. Acronyms and Abbreviations
❍ B. Definitions
❍ C. Design Principles

Figure 7.2 Outline of a software architecture document, as proposed by Kruchten (1995).

There can be little doubt that the '4 +1' Model and its successors have made a significant contribution to developing Software Architecture as a practical discipline. It does so in two particular ways when compared to the reductionist view of Software Architecture. Firstly, it extends the scope of architecture beyond mere structure. In Booch's UML World presentation already cited, one slide devoted to the domain of Software Architecture describes it not only in the traditional terms of the answers to the 'what?' and the 'how?' of system development, but also the 'why?' and the 'who?'. Booch (1999) offers a definition which, besides the usual structural stuff, adds 'Software architecture is not only concerned with structure and behaviour, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns' (p.458). Secondly, it places Software Architecture on the critical path of software development by insisting that the first prototype be an architectural one. In The Unified Software Development Process (Jacobson, 1999) this prototype is referred to as a 'small, skinny system'.

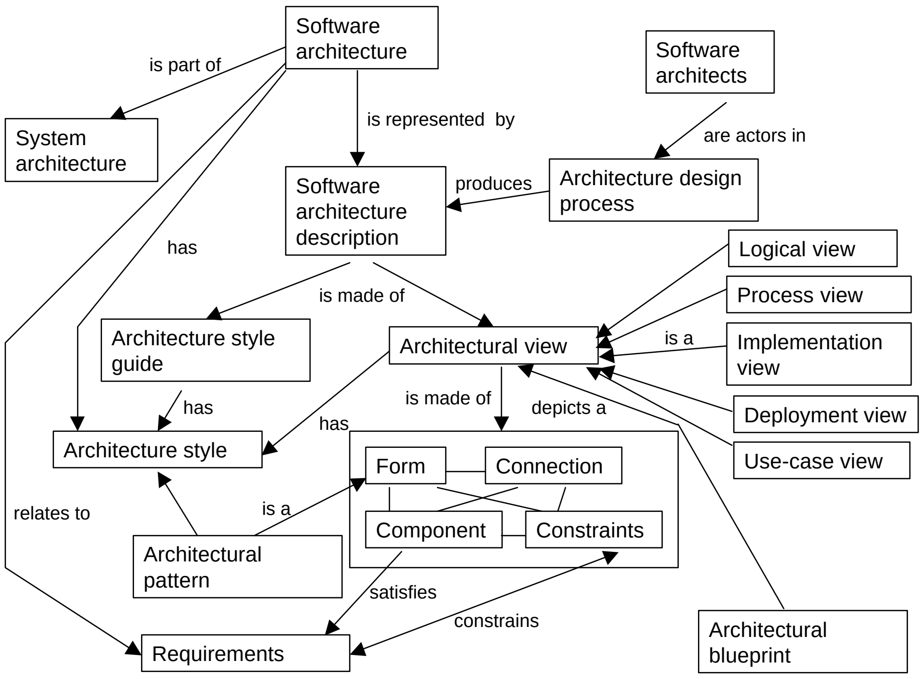


Figure 7.3 Booch's software architecture metamodel (adapted from Booch, 2000).

The '4+1 Model' and the Unified Process are designed to reveal the architecture as part of the software development process itself. In Kruchten's original paper he shows how the process works in domains as distinct as the Télic PABX architecture, and Hughes Aircraft's Air Traffic Control System but it less clear how it works when what is being created is an architecture for a domain, or a family of systems or a product-line system all of which require leverage from a common architecture. After all, if every project is free to discover the architecture then there is an accentuated possibility, if not a clear probability, that each such 'architecture' will be different.

Jan Bosch has recently published a contribution to the problem of adopting and evolving a product-line approach based on the work of the RISE (Research in Software Engineering) group at the University of Karlskrona/Ronneby in Sweden and its collaborations with companies such as Securitas Alarm AB and Axis Communications (Bosch, 2000). Product-line approaches stand close to those of Component-Based Development, a hot topic which grows hotter on an almost weekly basis it seems. The core of Bosch's approach involves first the development of what he calls *a functionality-based architectural design* from the requirements specification. At the centre of this activity is the search for key problem-driven architectural abstractions which he calls *archetypes*. Archetypes appear to be object types whose roots are in the problem space[1] but which, for product-line development, normally have to be generalized and abstracted further from the concrete phenomena that the analyst or architect first encounters. Bosch posits further that many archetypes are actually cross-domain in character.

A small and stable set of archetypes is chosen from a wider candidate list, often merging initial candidates to produce higher-level abstractions. The abstract relations between these archetypes are then identified and selected. The structure of the software architecture is then created by recursively decomposing the set of archetypes into well-defined components and establishing the relationship between them. A *system instantiation* can then be described in order to validate the work done thus far. This requires the further decomposition of components into lower-level components and the population of these components with instantiated archetypes. Since product-line architectures have to support a wide variety of systems with different specifications as well as different implementations, variability has to be addressed at this early point. Therefore, multiple such instantiations are produced to validate the match between the architecture and the requirements.

Architectural assessment involves another fairly novel idea: that of a *profile*. A profile is a set of scenarios, but they are not necessarily usage scenarios. That is to say, even if use cases are deployed in order to give traceability to functional requirements they will form only a subset of the overall number of scenarios described. Other scenarios, related to quality attributes (non-functional requirements), such as *hazard scenarios* (for safety critical systems) or *change scenarios* for flexible maintenance, are also specified. The profiles allow for a more precise specification of quality attributes than is typical in most software development. Bosch reports that between four and eight categories of scenarios are typical. Scenarios are identified and specified for each category, and each one

[1] The notion appears to be close to the notion captured in the ARCHETYPE pattern in the ADAPTOR pattern language (O'Callaghan, 2000b) but distinct from that of Peter Coad who uses the term to describe one of four meta-level collaborations he believes to be endemic in object systems (Coad, 1999).

assigned a weighting. Weightings are, if necessary, normalized. Scenario-based assessment follows in two main steps: impact analysis, and quality attribute prediction. The impact of the running of each scenario in a profile is assessed and then a predictive value for a quality attribute measured. So, for example, the number of changed and new components resulting from a change scenario in a maintenance category estimated in impact analysis could lead to a cost of maintenance effort estimation when brought together with a change frequency figure, based, say, on historical data in the second (predictive) phase. Profiles and scenarios can be used in a variety of ways, including simulated assessments in which both an executable version of the system and implemented profiles can be used to dynamically verify the architecture.

In the course of these assessments it is normal for one or more quality requirements to fail to be satisfied by the architecture, normally as a result of conflicting forces acting upon the overall design – for example, performance constraints being violated because of the levels of indirection built into the architecture in order to meet changeability requirements. In these circumstances what Bosch calls *architectural transformations* are required. Four categories of architectural transformation are identified and used in the following order, each next transformation having an expectancy of relatively decreasing impact:

- Impose architectural style (as per Shaw and Garlan, *op. cit.*);
- Impose architectural pattern (or mechanism) – by which Bosch means apply a rule locally to specify how the system will deal with one aspect of its overall functionality, e.g., concurrency or persistence[2];
- Apply design pattern (as per Gamma, 1995 – see below);
- Convert quality requirements to functionality – e.g. extend functionality to deal with self-monitoring, or redundancy for fault tolerance.

The final step is to distribute quality requirements to identified components or subsystems in the overall architecture. We restrict ourselves here to the discussion in Bosch's book to Software Architecture. A second and substantial part deals with the actual development of product lines. To the interested reader we recommend the purchase of the volume itself.

There is actually a fair amount in common between the '4+1' view and the approach to product-line development software architectures used by the RISE research group. Both pay due to the full scope of software architecture and deal with conceptual integrity, rationale as well as structure[3]. Both attach non-functional requirements to components or subsystems originally identified by analysing functionality. Both, at a certain point in the process, recommend the

---

2

application of architectural styles and then design patterns. We discuss issues involved therein below. The key difference is that Kruchten's approach is bottom-up, working from the abstractions discovered in a system requirements' specification, while Bosch's approach is decidedly top-down working from archetypal abstractions and then imposing them upon the development of a particular application or system.

7.2 Patterns, architecture and decoupled design

One of the most important recent ideas in software development is that of a design pattern. Design patterns are standard solutions to recurring problems, named to help people discuss them easily and to think about design. They have always been around in computing; so that terms such as 'linked list' or 'recursive descent' are readily understood by people in the field.

Software patterns have been described as reusable micro-architectures. Patterns are abstract, core solutions to problems that recur in different contexts but encountering the same 'forces' each time. The actual implementation of the solution varies with each application. Patterns are not, therefore, ready-made 'pluggable' solutions. They are most often represented in object-oriented development by commonly recurring arrangements of classes and the structural and dynamic connexions between them. Perhaps the best known and useful examples of patterns occur in application frameworks associated with graphical user interface building or other well-defined development problems. In fact, some of the motivation for the patterns movement came from the apprehension of already existing frameworks that led people to wonder how general the approach was. Nowadays it is more usual to deliver frameworks in the form of flexible class libraries for use by programmers in languages that support the class concept, often C++ and Java. Examples of frameworks range from class libraries that are delivered with programming environments through the NeXtStep Interface Builder to the many GUI and client/server development systems now on the market such as Delphi, Visual Studio, Visual Age or Visual Basic.

Patterns are most useful because they provide a language for designers to communicate in. Rather than having to explain a complex idea from scratch, the designer can just mention a pattern by name and everyone will know, at least roughly, what is meant. This is how designers in many other disciplines communicate their design ideas. In this sense they are an excellent vehicle for the collection and dissemination of the anecdotal and unquantifiable data that Borenstein (1991) argues needs to be collected before we can see real advances in the processes of building software. As with Software Architecture there are two different views of patterns abroad, both of which have value. To examine these we will first look at the roots of the patterns concept which lie outside the domain of software development. In fact they are to be found in the domain of the built environment. Hardly surprising then that patterns are closely related to software architecture.

Patterns are associated with the radical architect of the built environment, Christopher Alexander. From the outset of his career Christopher Alexander has been driven by the view that the vast majority of building stock created since the end of World War II (which constitutes the great majority of all construction works created by human beings in the history of the species) has been dehumanising, of poor quality and lacking all sense of beauty and human feeling. In his earliest publication Alexander presents a powerful critique of modern design (Alexander 1964) contrasting the failures of the professional *self-conscious* process of design with what he called the *unselfconscious* process by which peasants farmhouses, Eskimos' igloos and the huts of the Mousgoum tribesmen of the Cameroon amongst others create their living spaces. In the latter '…the pattern of building operation, the pattern of the building's maintenance, the constraints of the surrounding conditions, and also the pattern of daily life, are fused in the form' (p.31) yet there is no concept of 'design' or 'architecture', let alone separate designers and architects. Each man builds his own house.

Alexander argues that the unselfconscious process has an homeostatic (i.e. self-organising) structure that produces well-fitting forms even in the face of change, but in the self-conscious process this homeostatic structure has been broken down, making poorly-fitting forms almost inevitable[4]. Although, by definition, there are no explicitly articulated rules for building in the unselfconscious process there is usually a great weight of unspoken, unwritten, implicit rules that are, nevertheless, rigidly maintained by culture and tradition. These traditions provide a bedrock of stability, but more than that, a viscosity or resistance to all but the most urgent changes – usually when a form 'fails' in some way. When such changes are required the very simplicity of life itself, and the immediacy of the feedback (since the builder and homeowner are one and the same) mean that the necessary adaptation can itself be made immediately, as a 'one-off'. Thus the unselfconscious process is characterized by fast reactions to single 'failures' combined with resistance to all other changes. This allows the process to make a series of minor, incremental adjustments instead of spasmodic global ones. Changes have local impact only, and over a long period of time, the system adjusts 'subsystem by subsystem'. Since the minor changes happen at a faster rate of change than does the

---

[4] Mature biological systems are homeostatic. Consider how a tree, for example a mighty oak in a wood, is formed. The shape of an individual tree appears well adapted to its environment. The height of the tree is a factor of its competition with neighbouring trees. If used as a windbreak on the edges of farms, it will typically be bent in the direction of the prevailing wind patterns. The number of branches it has depends on the number of leaves it produces to accommodate local sunshine and rainfall conditions, etc. If alone on a hilltop, the pattern of growth is normally symmetrical, but if constrained in any way, the tree reflects the constraints in its own growth pattern. The tree's adaptiveness is of course a function of its genetic code. More recently Alexander has talked of his own approach as being a 'genetic' approach and the job of patterns is to instil this genetic code into structures.

culture, equilibrium is constantly and dynamically re-established after each disturbance.

In the self-conscious process tradition is weakened or becomes non-existent. The feedback loop is lengthened by the distance between the 'user' and the builder. Immediate reaction to failure is not possible because materials are not close to hand. Failures for all these reasons accumulate and require far more drastic action because they have to be dealt with in combination. All the factors which drive the construction process to equilibrium have disappeared in the self-conscious process. Equilibrium, if now reached at all, is unsustainable, not least because of the rate at which culture changes outpaces the rate at which adaptations can be made.

Alexander does not seek a return to primitive forms, but rather a new approach to a modern dilemma: the self-conscious designer, and indeed the notion of design itself, has arisen as a result of the increased complexity of requirements and sophistication of materials. She now has control over the process to a degree that the unselfconscious craftsman never had. But the more control she gets, the greater the cognitive burden and the greater the effort she spends in trying to deal with it, the more obscure becomes the causal structure of the problem which needs to be expressed for a well-fitting solution to be created. Increasingly, the very individuality of the designer is turning into its opposite: instead of being a solution, it is the main obstacle to a solution to the problem of restoring equilibrium between form and context.

In his 1964 work, Alexander produced a semi-algorithmic, mechanistic 'programme' based on functional decomposition (supported by a mathematical description in an appendix) to address the issues he identified. He has long since abandoned that prescription. It is the rather more informal drawings he used in the worked example which seem to have a more lasting significance. These became the basis, it seems, for the patterns in his later work.

Alexandrian 'theory' is currently expressed in an 11-volume strong literary project which does not include his 1964 work, *Notes on the Synthesis of Form*. Eight of these volumes have been published so far (though, at best, three of them, referred to as the patterns trilogy, *The Timeless Way of Building*, *A Pattern Language* and *The Oregon Experiment* are familiar to parts of the software patterns movement).[5] The ninth volume in the series, *The Nature of Order* is eagerly awaited as it promises to provide the fullest exposition yet of the underlying theory. A common theme of all the books is the rejection of abstract categories of architectural or design principles as being entirely arbitrary. Also rejected is the idea that it is even possible to successfully design 'very abstract forms at the big level'(Alexander 1996 p.8). For Alexander architecture gets its highest expression,

[5] *The Timeless Way of Building, A Pattern Language, The Oregon Experiment, The Linz Café, The Production of Houses, A New Theory of Urban Design, A Foreshadowing of 21st Century Art* and *The Mary Rose Museum* are all published by Oxford University Press. In preparation are *The Nature of Order*, *Sketches of a New Architecture* and *Battle: The Story of an Historic Clash Between World System A and World System B.*

not at the level of gross structure, but actually in its finest detail, what he calls ‘fine structure’. That is to say, the macroscopic clarity of design comes from a consistency, a geometric unity holds true at all levels of scale. It is not possible for a single mind to envision this recursive structure at all levels in advance of building it. It is in this context that his patterns for the built environment must be understood.

Alexander (1977) presents an archetypal pattern language for construction. The language is an interconnected network of 253 patterns that encapsulate design best practice at a variety of levels of scale, from the siting of alcoves to the construction of towns and cities. The language is designed to be used collaboratively by all the stakeholders in a development, not just developers. This is premised, in part at least, by the idea that the real experts in buildings are those that live and work in them rather than those that have formally studied architecture or structural engineering. The patterns are applied sequentially to the construction itself. Each state change caused by the application of a pattern creates a new context to which the next pattern can be applied. The overall development is an emergent property of the application of the pattern language. The language therefore has a generative character: it generates solutions piece-meal from the successive addressing of each individual problem that each of the patterns addresses separately.

WAIST-HIGH SHELF (number 201 in the language) is an example pattern. It proposes the building of waist-high shelves around main rooms to hold the ‘traffic’ of objects that are handled most so that they are always immediately at hand. Clearly the specific form, depth, position and so on of these shelves will differ from house to house and workplace to workplace. The implementation of the pattern creates, therefore, a very specific context in which other patterns such as THICKENING THE OUTER WALL (number 211) can be used since Alexander suggests the shelves be built into the very structure of the building where appropriate, and THINGS FROM YOUR LIFE (number 253) to populate the shelves.

The pattern which more than any other is physical and procedural embodiment of Alexander’s approach to design, however, is pattern number 208 GRADUAL STIFFENING:

> ‘The fundamental philosophy behind the use of pattern languages is that buildings should be uniquely adapted to individual needs and sites; and that the plans of buildings should be rather loose and fluid, in order to accommodate these subtleties….
>
> ‘Recognize that you are not assembling a building from components like an erector set, but that you are instead weaving a structure which starts out globally complete, but flimsy; then gradually making it stiffer but still rather flimsy; and only finally making it completely stiff and strong.’ (Alexander, 1977, p.963-969).

In the description of this pattern Alexander invites the reader to visualize a 50-year old master carpenter at work. He keeps working, apparently without stopping, until he eventually produces a quality product. The smoothness of his labour comes from the fact that he is making small, sequential, incremental steps such that he can always eliminate a mistake or correct an imperfection with the next step. He compares this with the novice who with a 'panic-stricken attention to detail' tries to work out everything in advance, fearful of making an unrecoverable error. Alexander's point is that most modern architecture has the character of the novice's work, not the master craftsman's. Successful construction processes, producing well-fitting forms, comes from the postponement of detail design decisions until the building process itself so that the details are fitted into the overall, evolving structure.

Alexander's ideas seem to have been first introduced into the object-oriented community by Kent Beck and Ward Cunningham. In a 1993 article in *Smalltalk Report* Beck claimed to have been using patterns for six years already, but the software patterns movement seems to have been kicked off by a workshop on the production of a software architect's handbook organized by Bruce Anderson for OOPSLA '91. Here met for the first time Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides – a group destined to gain notoriety as the Gang of Four (GoF). Gamma was already near to completion of his PhD thesis on 'design patterns' in the ET++ framework (Gamma 1991,1992). He had already been joined by Richard Helm in the production of an independent catalogue. By the time of a follow-up meeting at OOPSLA in 1992 first Vlissides and then Johnson had joined the effort and, sometime in 1993 the group agreed to write a book that has been a best-seller ever since its publication in 1995. In fact, outside of the patterns movement itself many in the software development industry identify software patterns completely and totally with the GoF book.

However the 1991 OOPSLA workshop was only the first in a series of meetings that culminated first in the formation of the non-profit Hillside Group[6] (apparently so-called because they went off to a hillside one weekend to try out Alexander's building patterns) and then the first Pattern Languages of Programming (PLoP) conference in 1994. PLoP conferences, organized and funded by the Hillside Group, take place annually in America, Germany and Australia and collections of the patterns that are produced are published in a series by Addison Wesley- four volumes to date. In addition the Hillside Group maintains a web site and numerous pattern mailing lists (….Hillside, 2000). These communication channels form the backbone of a large and growing community that is rightly called the patterns movement.

A characteristic of the way patterns are developed for publication in the patterns movement is the so-called pattern writers' workshop. This is a form of peer-review which is loosely-related to design reviews that are typical in software development processes, but more strongly related to poetry circles which are decidedly atypical. The special rules of the pattern writers workshop (which are the

[6] The founding members were Ken Auer, Kent Beck, Grady Booch, Jim Coplien, Ward Cunningham, Hal Hildebrand and Ralph Johnson. The initial sponsors were Rational and the Object Management Group.

modus operandi of the PLoP conferences) have been shown to be powerful in producing software patterns, written in easily accessible, regular forms known as **pattern templates**, at an appropriate level of abstraction. Linda Rising reports on their effectiveness in producing a patterns' culture in the telecommunications company AGCS (Rising 1998), and they are *de rigeur* in parts of IBM, Siemens and AT&T, all of which are known to have produce their own in-house software patterns, as well as published them in the public domain.

While the GoF book has won deserved recognition for raising the profile of patterns, for many it has been a double-edged sword. The GoF patterns form a catalogue of standalone patterns all at a similar level of abstraction. Such a catalogue can never have the generative quality that Alexander's pattern language claims for itself and, to be fair, the Gang of Four freely admit that this was not the aim of their work.

The GoF book includes 23 useful design patterns, including the following particularly interesting and useful ones:

- FAÇADE. Useful for implementing object wrappers.
- ADAPTER.
- PROXY.
- OBSERVER. This helps an object to notify registrants that its state has changed and helps with the implementation of blackboard systems.
- VISITOR and STATE. These two patterns help to implement dynamic classification.
- COMPOSITE.
- BRIDGE. Helps with decoupling interfaces from their implementations.

Some cynics claim that some of the GoF patterns are really only useful for fixing deficiencies in the C++ language. Examples of these might arguably include DECORATOR and ITERATOR. However, the very suggestion raises the issue of language-dependent vs. language-independent patterns. Buschmann *et al*. (1996) (also known as the Party of Five or PoV) from Siemens in Germany suggest a system of patterns that can be divided into architectural patterns, design patterns and language idioms. They present examples of the first two categories. Architectural patterns include: PIPES AND FILTERS, BLACKBOARD systems, and the MODEL VIEW CONTROLLER (MVC) pattern for user interface development. Typical PoV design patterns are called:

- FORWARDER RECEIVER;
- WHOLE PART; and
- PROXY.

The reader is advised by the PoV to refer to all the GoF patterns as well. The PoV book can therefore be regarded as an expansion of the original catalogue, not merely through the addition of extra patterns, but by addressing different levels of abstraction too. The whole-part pattern is exactly the implementation of the composition structures that form part of basic object modelling semantics. In that sense it appears to be a trivial pattern. However, since most languages do not support the construct, it can be useful to see the standard way to implement it. It is a rare example of a design pattern that maps directly to an idiom in several languages: a multi-language idiom. The most well-known source of idiomatic (i.e., language-specific) patterns is Jim Coplien's book on advanced C++ which predates which is itself a list, and a program is made up of primitive statements and blocks that are themselves made of blocks and statements. We can use a Catalysis framework template to document a general case as in Figure 7.4, in which one might substitute Block for <Node>, Program for <Branch> and Statement for for <Leaf>.

We process recursive structures by recursive descent and need to specify – in the template – whether looping is prevented by including constraints. In OCL such a constraint can be written:

```
context Node::ancestors = parent + parent.ancestors AND
context Node:: not(ancestors includes self)
```

The list and the block patterns can be regarded as whole objects or wrappers and the pattern merely describes their internal structure.

The above examples indicate that a standard pattern layout may be beneficial, and many proponents adopt a standard based on Alexander's work: the so-called Alexandrian form. This divides pattern descriptions into prose sections with suitable pictorial illustrations as follows; although the actual headings vary from author to author.

- Pattern name and description.
- Context (Problem) – situations where the patterns may be useful and the problem that the pattern solves.
- Forces – the contradictory forces at work that the designer must balance.
- Solution – the principles underlying the pattern and how to apply it (including examples of its realization, consequences and benefits).
- Also Known As/Related patterns – other names for (almost) the same thing and patterns that this one uses or might occur with.
- Known uses.

Kent Beck has produced a book of 92 Smalltalk idioms (Beck, 1997) and there have been a number of language-specific 'versions' of the GoF book, notably for Smalltalk (Alpert *et al*., 1998) and Java (Grand 1998, Cooper 2000). Although many of the PoV architectural 'patterns' exist also among the SEI's 'styles', it is crucial to note their different purposes. Both are abstracted from software development best practice, but by the SEI in order to collect and formalize (and presumably later automate) them, by the PoV in order to further generalize that practice.

The overwhelming majority of software patterns produced to date have been design patterns at various levels of abstraction but Fowler (1997) introduces the idea of analysis patterns as opposed to design patterns. Fowler's patterns are reusable fragments of object-oriented specification models made generic enough to be applicable across a number of specific application domains. They therefore have something of the flavour of the GoF pattern catalogue (described in that book's subtitle as 'elements of reusable object-oriented software') but are even further removed from Alexander's generative concepts. Examples of Fowler's patterns include:

- PARTY: how to store the name and address of someone or something you deal with.
- ORGANIZATION STRUCTURE: how to represent divisional structure.
- POSTING RULES: how to represent basic bookkeeping rules.
- QUOTE: dealing with the different way financial instrument prices are represented.

There are many more, some specialized into domains such as Health Care or Accounting.

The problem with these patterns is that even the simplest ones – like ACCOUNTABILITY – are really quite hard to understand compared to the difficulty of the underlying problem that they solve. My experience was that it took three attempts at reading the text before really understanding what was going on. At the end of this process I found that I knew the proposed solution already but would never have expressed it in the same terms.

Maiden *et al*. (1998) propose a pattern language for socio-technical system design to inform requirements validation thereof, based on the CREWS-SAVRE prototype discussed in Chapter 8. They specify three patterns as follows:

- MACHINE-FUNCTION: this represents a rule connecting the presence of a user action (a task script in our language) to a system requirement to support that action (an operation of a business object that implements the task). I feel that it is stretching language somewhat to call this rule a pattern.
- COLLECT-FIRST-OBJECTIVE-LAST: this pattern tells us to force the user to complete the prime transaction after the subsidiary ones; e.g. ATMs should make you take the card before the cash. (For a discussion of the psychological phenomenon of *completion* in user interface design, see Graham, 1995.)
- INSECURE-SECURE-TRANSACTION: this suggests that systems should monitor their security state and take appropriate action if the system becomes insecure.

The value of these patterns may be doubted because, like Fowler‘s analysis patterns, they seem to state the obvious; and they fail to address the sort of task or system usage patterns represented by our task association sets or use case refinement. Also it could be argued that they are nothing but design principles; just as *completion* provides a well-known design principle in HCI. On the other hand their operationalization in the CREWS-SAVRE system indicates that they may have a specialized practical value in this and certain other contexts.

OTHER PATTERN TYPES

There has also been interest in developing patterns for organizational development (Coplien, 1995; O'Callaghan, 1997, 1997a, 1998). Coplien applies the idea of patterns to the software development process itself and observes several noteworthy regularities. These observations arose out of a research project sponsored by AT&T investigating the value of QA process standards such as ISO9001 and the SEI's Capability Maturity Model. Working from a base concept that real processes were characterized by their communication pathways. Coplien, together with Brad Cain and Neil Harrison analysed more than 50 projects by medium-size, high productivity software development organizations including the Borland team charged with developing the Quattro Pro spreadsheet product (Cain, ….). The technique they used was to adapt CRC cards and to get, in a workshop situation, representatives of the development organization under focus to enumerate roles (as opposed to job descriptions), identify and categorize the strength of the relationships between those roles (as either Weak, Medium and Strong) and then to rôle-play the development process in order to validate their judgements. The information was then input into a Smalltalk-based system called Pasteur that produces a variety of different sociometric diagrams and measures. From these Coplien *et. al*. were able to identify the commonly recurring key characteristics of the most productive organizations and develop a 42-strong pattern language to aid the design of development organizations. Included in the language are such patterns as:

- CONWAY'S LAW states that architecture always follows organization or *vice versa*;
- ARCHITECT ALSO IMPLEMENTS requires that the architect stands close to the development process;
- DEVELOPER CONTROLS PROCESS requires that the developer's own and drive the development process, as opposed to having one imposed on them;
- MERCENARY ANALYST enables the 'off-line' reverse engineering and production of project documentation;
- FIREWALL describes how to insulate developers from the 'white noise' of the software development industry;
- GATEKEEPER describes how to get useful information in a timely manner to software developers.

A typical application of such organizational patterns is the combined use of GATEKEEPER and FIREWALL in, say, a situation where a pilot project is assessing new technology. The software development industry excels at rumour-mongering, a situation fuelled by the practice of vendors who make vapourware announcements long in advance of any commercial-strength implementations. Over-attention to the whispers on the industry grapevine, let alone authoritative-looking statements in the trade press, can seriously undermine a pilot project. Developers lose confidence in Java, say, because of its reputation for poor performance or a claimed lack of available tools. Yet, at the same time, some news is important: for example, the publication of Platform 2 for Java. A solution is to build official firewalls and then create a gatekeeper role where a nominated individual, or perhaps a virtual centre such as an Object Centre, is responsible for filtering and forwarding the useful and usable information as opposed to unsubstantiated scare stories, junk mail and even the attention of vendors' sales forces.

More interesting than the individual patterns themselves, however, is the underlying approach of Coplien's language which is much closer to the spirit of Alexander's work than anything to be found in the GoF or PoV books, for example. First, since its very scope is intercommunication between people, it is human-centred. Second, it is explicitly generative in its aim. Coplien argues that while software developers do not inhabit code in the way that people inhabit houses and offices, as professionals they are expert users of professional processes and organizations. Therefore, just as Alexander's language is designed to involve all the stakeholders of building projects (and, above all, the expertise of the users of the buildings) so process designers have to base themselves on the expertise of the victims of formal processes – the developers themselves. Coplien's attempt to create an avowedly Alexandrian pattern language seems to push the focus of his patterns away from descriptions of fragments of structure (as is typical in the GoF patterns) much more towards descriptions of the work that has to be done. In going beyond mere structure Coplien's patterns have much more of a feel of genuine architecture about them than do many other pattern types available.

In fact it is clear that from common roots there are two polarised views of patterns abroad today. One view focuses on patterns as generic structural descriptions. They have been described, in UML books especially, as 'parameterized collaborations'. The suggestion is that you can take, say, the structural descriptions of the roles that different classes can play in a pattern and then, simply by changing the class names and providing detailed algorithmic implementations, plug them into a software development. Patterns thus become reduced to abstract descriptions of potentially pluggable components. A problem with this simplistic view occurs when a single class is required to play many roles simultaneously in different patterns. Erich Gamma has recently re-implemented the HotDraw framework, for example, in Java. One class, Figure, appears to collaborate in fourteen different overlapping patterns – it is difficult to see how the design could have been successful if each of these patterns had been instantiated as a separate component. More importantly, this view has nothing to say about how software projects should be put together, only what (fragments of) it might look like structurally. The other view regards them simply as design decisions (taken in a particular context, in response to a problem recognized as a recurring one). This view inevitably tends toward the development of patterns as elements in a generative pattern language.

Support for this comes from the work of Alan O'Callaghan and his colleagues in the Object Engineering and Migration group and Software Technology Research Laboratory at De Montfort University. O'Callaghan is the lead author of the ADAPTOR pattern language for migrating legacy systems to object and component-based structures. ADAPTOR was based initially on five projects, starting in 1993, in separate business areas and stands for Architecture-Driven And Patterns-based

Techniques for Object Re-engineering. It currently encapsulates experiences of eight major industrial projects in four different sectors: telecommunications, the retail industry, defence and oil exploration. O'Callaghan argues that migrating to object technology is more than mere reverse engineering, because reverse engineering is usually (a) formal and (b) focused purely on the functional nature of the legacy systems in question and (c) assumes a self-similar architecture to the original one. The most crucial information, about the original design rationales, has already been lost irretrievably. It cannot be retrieved from the code because the code never contained that information (unless, of course, it was written in an unusually expressive way). The best that traditional archaeological approaches to reverse engineering can achieve is to recreate the old system in an object-oriented 'style' which, more often than not, delivers none of the required benefits.

The approach pioneered by O'Callaghan was to develop object models of the required 'new' system the legacy system, and by focusing on the maintainers and developers (including their organizational structure) rather than the code or design documentation, determine only subsequently what software assets might already exist that could be redeployed. His group turned to patterns in the search for some way of documenting and communicating the common practices that were successful in each new project (legacy systems present especially wicked problems are, overall, always unique unto themselves). At first public domain, standalone design patterns were used but quickly his team were forced to mine their own. Then problems of code ownership (i.e. responsibility for part of a system being re-engineered belonging to someone other than the immediate client) caused by the fact that migrations typically involve radical changes at the level of the gross structure of a system, required that organizational and process problems be addressed also through patterns. Finally, observations that the most powerful patterns in different domains were interconnected suggested the possibility of a generative pattern language.

ADAPTOR was announced in 1998 as a 'candidate, open, generative pattern language'. It is a candidate language for two reasons: first, despite the overwhelming success of the projects from which it is drawn ADAPTOR is not comprehensive enough in its coverage or recursed to a sufficient level of detail to be, as yet, truly generative. Secondly, O'Callaghan has different level of confidence in the different patterns with only those having gone through the patterns workshops of the patterns movement being regarded as fully mature. Patterns yet to prove themselves in this way are regarded as candidate patterns. ADAPTOR is open in a number of senses too. First, like any true language both the language itself and the elements which comprise it are evolvable. Many of the most mature patterns, such as GET THE MODEL FROM THE PEOPLE which was first presented in 1996 at a TelePlop workshop, have gone through numbers of iterations of change. Secondly, following Alexander (1977), O'Callaghan insists that patterns are open abstractions themselves. Since no true pattern provides a complete solution and every time it is applied it delivers different results (because of different specific contexts to which it is applied), it resists the kind of formalization that closed abstractions such as rules can be subject to. Finally, and uniquely amongst published software pattern languages ADAPTOR is open because it makes explicit use of other public-domain pattern languages and catalogues, such as Coplien's generative development-process language already cited, or the GoF and PoV catalogues.

Patterns in ADAPTOR include:

- GET THE MODEL FROM THE PEOPLE requires utilization of the maintainers of a legacy system as sources of business information;
- PAY ATTENTION TO THE FOLKLORE treats the development/maintenance communities as domain experts, even if they don't do so themselves;
- BUFFER THE SYSTEM W ITH SCENARIOS gets the main business analysts, marketeers, futurists, etc. to rôle-play alternative business contexts to the one they bet on in their requirements specifications;
- SHAMROCK divides a system under development into three loosely coupled 'leaves' – each of which could contain many class categories or packages – the leaves are the conceptual domain (the problem space objects), the infrastructure domain (persistence, concurrency etc.) and the interaction domain (GUI's, inter-system protocols etc.);
- TIME-ORDERED COUPLING clusters classes according to common change rates to accommodate flexibility to change;
- KEEPER OF THE FLAME sets up a role whereby the detailed design decisions can be assured to be in continuity with the architecture. Permits changes to the gross structure if deemed necessary and appropriate;
- ARCHETYPE creates object types to represent the key abstractions discovered in the problem space;
- SEMANTIC WRAPPER creates wrappers for legacy code that presents behavioural interfaces of identifiable abstractions to the rest of the system.

Something of the open and generative character aspired to by ADAPTOR can be gained from looking at the typical application of patterns to the early phases of a legacy system migration project. Underpinning ADAPTOR is the model-driven approach described earlier. O'Callaghan's problem space models are comprised of object types and the relationships between them which capture the behaviour of key abstractions of the context of the system as well as the system itself. ARCHETYPE is therefore one of the first patterns used, along with GET THE MODEL FROM THE PEOPLE and PAY ATTENTION TO THE FOLKLORE. At an early stage strategic 'what-if' scenarios are run against this model using BUFFER THE SYSTEM WITH SCENARIOS. SHAMROCK is applied in order to decouple the concept domain object types from the purely system resources needed to deliver them at run-time. The concept domain 'leaf' can then be factored into packages using TIME-ORDERED COUPLING to keep types with similar change rates (discovered through the scenario-buffering) together. Coplien's CONWAY'S LAW is now utilized to design a development organization that is aligned with the evolving structure of the system. CODE OWNERSHIP (another Coplien pattern) makes sure that every package has someone assigned to it with responsibility for it. An ADAPTOR pattern called TRACKABLE COMPONENT ensures that these 'code owners' are responsible for publishing the interfaces of their packages that others need to develop to so that they can evolve in a controlled way. The GoF pattern FAÇADE is deployed to create a scaffolding for the detailed structure of the system. It is at this point that decisions can be made as to which pieces of functionality require new code and which can make use of legacy code. The scaffolding ensures that these decisions, and their implementation consequences, can be dealt with at a rate completely under the control and at the discretion of the development team without fear of runaway ripple effects. For the latter, SEMANTIC WRAPPERS are used to interface the old legacy stuff to the new object-oriented bits.

Even with this cursory example we can see how the language addresses all of the important issues of architecture (client's needs, conceptual integrity, structure, process and organization etc.) as well as getting quickly to the heart of the issues of legacy migration. O'Callaghan reports that, when outlining this approach at a public tutorial, one member of the audience objected that the model-driven approach was not re-engineering at all but just 'forward engineering with the reuse of some legacy code' (O'Callaghan, personal communication with Ian Graham). In reply, O'Callaghan agreed and stated that that was just the point. On further consideration, he decided that many of ADAPTOR's patterns were not specific to legacy migration at all. As a result ADAPTOR is currently being regarded as a subset of a more general language on architectural praxis for software development in a project codenamed the Janus project (O'Callaghan, 2000).

The debate about the nature of software patterns ('parameterized collaborations' *versus* 'design decisions'); pattern catalogues *versus* pattern languages) itself both reflects, and impacts upon, the debates about software architecture discussed in the previous section. That relationship has been sharply exposed by Coplien's guest editorship of *IEEE Software* magazine in the Autumn of 1999. The issue was a special feature on software architecture in which Coplien published, amongst others, Alexander's keynote talk to the OOPSLA conference in San Jose, California in 1996 (Alexander, 1999). In his editorial, re-evaluating the architectural metaphor, Coplien identified two fundamental approaches to software development: the 'blueprint' or 'masterplan' approach *versus* that of 'piecemeal growth' (Coplien, 1999). Coplien suggests that the immature discipline of software architecture is suffering from 'formal envy' and has borrowed inappropriate lessons from both the worlds of hardware engineering and the built environment. Symptoms of its crisis are the separation of the deliverables of architecture from the artefacts delivered to the customer and the reification of architecture as a separate process in a waterfall approach to software development. Following the architect of the built environment Ludwig Miles van der Rohe, Coplien proclaims, as does Alexander as we have seen, that 'God lives in the details' and that clarity at the macro level can only be judged by whether it incorporates the fine details successfully. He further asserts: 'The object experience highlights what had been important all along: architecture is not so much about software, but about the people who write the software' (p.41).

The main point about coupling and cohesion is that it permits people to work socially to produce a piece of software and both recognize and value their own particular contribution. Coplien points to CRC cards and their use in object-oriented development as the classic example of software design's anthropomorphic nature. From this perspective, software patterns were the next wave in the advance of a software architectural practice of this kind. As Coplien quite rightly points out, the patterns movement has always celebrated the otherwise lowly programmer as the major source of architectural knowledge in software development. Beyond that it recognizes the deep character of the relationship between code's structure and the communication pathways between the people developing and maintaining it. In doing so, Coplien argues, patterns have taken software development beyond the naïve practice of the early days of objects, which fell short of its promise because it was still constrained by a purely modular view of software programs, inherited from the previous culture. Further advance requires liberation from the weight of 'the historic illusions of formalism and planning'(p.42).

Richard Gabriel, who is currently a member of the Hillside Group, a master software practitioner as well as a practising poet[7], suggests that there are two reasons why all successful software development is in reality piecemeal growth. First, there is the cognitive complexity of dealing not only with current, but possible future, causes of change which make it impossible to completely visualize a constructible software program in advance to the necessary level of detail with any accuracy (Gabriel, 1996). Second, there is the fact that pre-planning alienates all but the planners. Coplien, Gabriel and the entire patterns movement are dedicated to developing practices which combat this social alienation. In doing so they impart a profound social and moral obligation to the notion of software architecture. In the face of these stark realities the only alternative to piecemeal growth is the one offered by Parnas (1986): to fake the blueprints by reverse engineering them once the code is complete.

7.2.1 Design patterns for decoupling

To get the true benefits of polymorphism – or 'pluggability' – in a program, it is important to declare variables and parameters not with explicit classes, but *via* an interface (abstract class in C++, interface in Java, deferred class in Eiffel). Looking at the metaphor of a café used in Figure 7.5, our initial model has Food that are not edible – we could start selling newspapers in our café. Our original Food class happens to implement both interfaces. Some good programmers insist that we should always use interfaces to declare variables and parameters. Simple multiple inheritance of rôle-types can be considered a pattern for composing collaborations. (In untyped languages such as Smalltalk, the difference appears only in our design model, and doesn't appear in the program.)

Using interfaces is the basic pattern for reducing dependencies between classes. A class represents an implementation; an interface represents the specification of what a particular client requires. So declaring an interface pares the client's dependency on others down to the minimum: anything will do, that meets the specification represented by the interface.

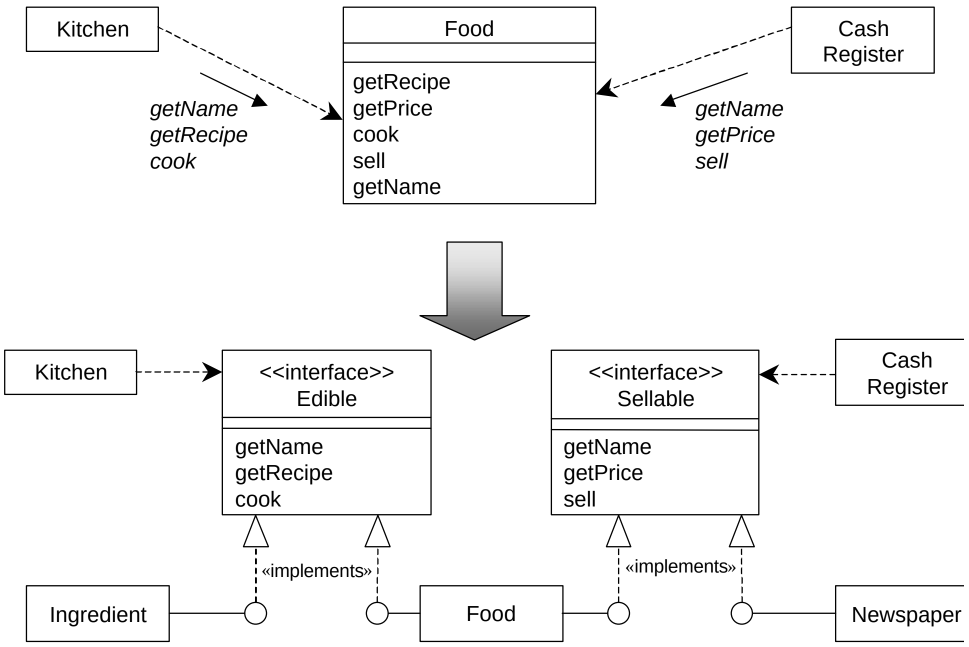


Figure 7.5 Interface decoupling

DECOUPLING WITH FACTORY

When we add a new class to a program, we want to alter code at as few places as possible. Therefore, one should minimize the number of points where classes are explicitly mentioned. As we have discussed, some of these points are in variable and parameter declarations: use interface names there instead. The other variation points are where new instances are created. An explicit choice of class has to be made somewhere (using new Classname, or in some languages, by cloning an existing object).

FACTORY patterns are used to reduce this kind of dependency. We concentrate all creations into a **factory**. The factory's responsibility is to know what classes there are, and which one should be chosen in a particular case. The factory might be a method, or an object (or possibly one rôle of an object that has associated responsibilities). As a simple example, in a graphical editor, the user might create new shapes by typing 'c' for a new circle, 'r' for a new rectangle and so on. Somewhere we must map from keystrokes to classes, perhaps using a switch statement or in a table. We do that in the shape factory, which will typically have a method called something like make ShapeFor(char keystroke). Then, if we change the design to permit a new kind of shape, we would add the new class as a subclass of Shapes and alter the factory. More typically, there will be a menu that initializes from a table of icons and names of shape types.

Normally one would have a separate factory for each variable feature of the design: one factory for creating shapes and a separate one for creating pointing-device handlers.

A separate factory class can have subclasses. Different subclasses can provide different responses as to which classes should be created. For example, suppose we permit the user of a drawing editor to change mode between creating plain shapes and creating decorated ones. We add new classes like FancyTriangles to accompany the plain ones but, instead of providing new keystrokes for creating them explicitly, we provide a mode switch – whose effect is to alter this pointer in the Editor:

ShapeFactory shapeFactory;

Normally, this points to an instance of PlainShapeFactory, which implements ShapeFactory. When given the keystroke 'c' or the appropriate file segment, this factory creates a normal circle. But we can reassign the pointer to an instance of FancyShapeFactory, which, given the same inputs, creates a FancyCircle. Gamma *et al.* (1995) call classes like ShapeFactory **abstract factories**. It declares all the factory messages like makeShapeFor(keystroke), but its subclasses create different versions.

DECOUPLING WITH DELEGATION

Programmers new to object oriented design can get over-enthusiastic about inheritance. A naïve analysis of our hotel system example might conclude that there are several kinds of hotel, which allocate rooms to guests in different ways; and a naïve designer might therefore create a corresponding set of subclasses of Hotel in the program code, overriding the room-allocation method in the different subclasses:

```
class Hotel {...
   public void checkInGuest(...)
   ...
   abstract protected Room allocateRoom (...);
   ...}
class LeastUsedRoomAllocatingHotel extends Hotel
{  protected Room allocateRoom (...)
   {     // allocate least recently used room
   ...}
```

```
}
class EvenlySpacedRoomAllocatingHotel extends Hotel
{  protected Room allocateRoom (...)
   { // allocate room furthest from other occupied
```

This is not a very satisfactory tactic: it cannot be repeated for other variations in requirements: if there are several ways of paying the staff for example. Overriding methods is one of the shortest blind alleys in the history of object-oriented programming. In practice, it is useful only within the context of a few particular patterns (usually those concerned with providing default behaviour). Instead, the trick is to move each separate behavioural variation into a separate object. We define a new interface for room allocation, for staff payment, and so on; and then define various concrete implementations for them. For example, we might write:

```
class Hotel {
   Allocator allocator; ...
   public void checkInGuest (...)
   {... allocator.doAllocation(...);..}
...}
interface Allocator{
   Room doAllocation (...); // returns a free room
...}
class LeastUsedAllocator implements Allocator
{  Room doAllocation (...) {...code ...}}
class EvenSpaceAllocator implements Allocator
{  Room doAllocation (...) {...code ...}}
```

This pattern of moving behaviour into another object is called DELEGATION. It has some variants, described differently depending on your purpose. One benefit of delegation is that it's possible to change the room allocator (for example) at run time, by 'plugging in' a new Allocator implementation to the allocator variable. Where the objective is to do this frequently, the pattern is called STATE.

Another application DELEGATION is called POLICY: this separates business dependent routines from the core code; so that it is easy to change them. Room allocation is an example. Another style of POLICY checks, after each operation on an object, that certain business-defined constraints are matched, raising an exception and cancelling the operation if not. For example, the manager of a hotel in a very traditional region of the world might wish to ensure that young persons of opposite gender are never assigned rooms next to each other: the rule would need to be checked whenever any room-assigning operation is done.

DECOUPLING WITH EVENT, OBSERVER AND MVC

Interfaces decouple a class from explicit knowledge of how other objects are implemented; but in general there is still some knowledge of what the other object does. For example, the RoomAllocator interface includes allocateRoom(guest) – it is clear what the Hotel expects from any RoomAllocator implementation. But sometimes it is appropriate to take decoupling a stage further, so that the sender of a message does not even know what the message will do. For example, the hotel object could send a message to interested parties whenever a room is assigned or becomes free. We could invent various classes to do something with that information: a counter that tells us the current occupancy of a room, a reservations system, an object that directs the cleaning staff, and so on.

These messages are called **events**. An event conveys information; unlike the normal idea of an operation, the sender has no particular expectations about what it will do; that is up to the receiver. The sender of an event is designed to be able to send it to other parties that register their interest; but it does not have to know anything about their design. Events are a very common example of decoupling.

To be able to send events, an object has to provide an operation whereby a client can register its interest; and it has to be able to keep a list of interested parties. Whenever the relevant event occurs, it should send a standard notification message to each party on the list.

An extension of the EVENT pattern is OBSERVER. Here the sender and listener are called Subject and Observer, and an event is sent whenever a change occurs in the sender's state. By this means, the observers are kept up to date with changes in the subject. New observers may be added easily as subtypes as shown in Figure 7.6.

A very common application of OBSERVER is in user interfaces: the display on the screen is kept up to date with changes in the underlying business objects. Two great benefits this usage are:

1. the user interface can easily be changed without affecting the business logic;
2. several views of a business object can be in existence at a time – perhaps different classes of view.

For example, a machine design can be displayed both as an engineering drawing and as a bill of materials; any changes made *via* one view are immediately reflected in the other. Users of word processors and operating systems are also familiar with changes made in one place – perhaps to a file name – appearing in another. It is only very old technology in which a view needs to be prompted manually to reflect changes made elsewhere. Another common use of OBSERVER is in blackboard systems.

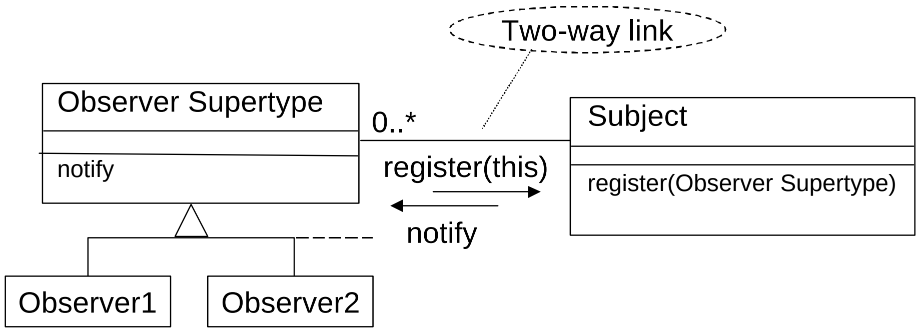


Figure 7.6 OBSERVER

The OBSERVER pattern has its origin in the MODEL-VIEW-CONTROLLER (MVC) pattern (or 'paradigm' as it is often mistakenly called), first seen in the work of Trygve Reenskaug on Smalltalk in the 1970s, and now visible in the Java AWT and Swing libraries. The MVC metaphor also influenced several object-oriented and object-based visual programming languages such as Delphi and Visual Basic.

An MVC **model** is an object in the business part of the program logic: not to be confused with our other use of the term 'modelling'. A **view** is an observer whose job it is to display the current state of the model on the screen, or whatever output device is in use: keeping up to date with any changes that occur. In other words, it translates from the internal representation of the model to the human-readable representation on the screen. **Controller** objects do the opposite: they take human actions, keystrokes and mouse movements, and translate them into operations that the model object can understand. Note, in Figure 7.7, that OBSERVER sets up two-way visibility between model and controller as well as model and view. Controllers are not visible to views, although part of a view may sometimes be used as a controller: cells in a spreadsheet are an example.

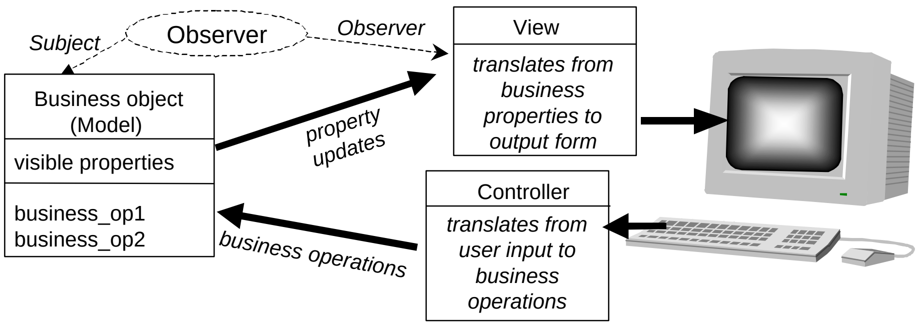


Figure 7.7 MODEL-VIEW-CONTROLLER instantiates OBSERVER

Views are often nested, since they represent complex model objects, that form whole-part hierarchies with others. Each class of controller is usually used with a particular class of views, since the interpretation of the user's gestures and typing is usually dependent on what view the mouse is pointing at.

DECOUPLING WITH ADAPTER

In MVC, View and Controller are each specializations of the ADAPTER pattern (not to be confused with the ADAPTOR pattern language). An **adapter** is an object that connects two others of classes that were designed in ignorance of each other, translating events issued by one into operations on the other. The View translates from the property change events of the Model object to the graphical operations of the windowing system concerned. A Controller translates from the input events of the user's gestures and typing to the operations of the Model.

An adapter knows about both of the objects it is translating between; the benefit that it confers is that neither of them needs to know about the other, as we can see from the package dependency diagram in Figure 7.8.

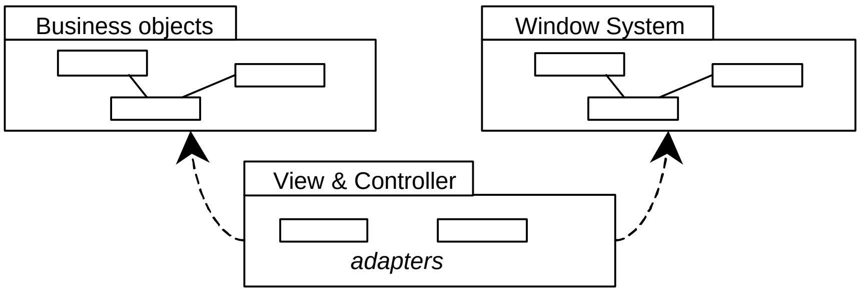


Figure 7.8 Adapters decouple the objects they connect

Any collection of functions can be thought of as an adapter in a general sense; but the term is usually used in the context of event; that is, where the sender of a message does not know what is going to be done with it.

Adapters appear in a variety of other contexts beside user interfaces; and also on a grander scale: they can connect components running in separate execution spaces. Adapters are useful as 'glue' wherever two or more pre-existing or independently-designed pieces of software are to be made to work together.

DECOUPLING WITH PORTS AND CONNECTORS

Adapters generally translate between two specific types – for example, from the keystrokes and mouse clicks of the GUI to the business operations of a business object. This means, of course, that one has to design a new adapter whenever one wants to connect members of a different pair of classes. Frequently, that is appropriate: different business objects need different user interfaces. But it is an attractive option to be able to reconfigure a collection of components without designing a new set of adapters every time.

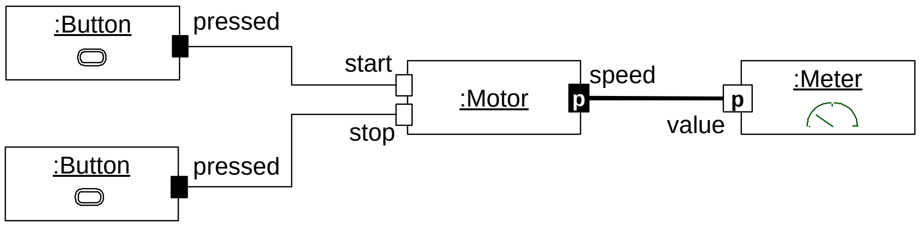
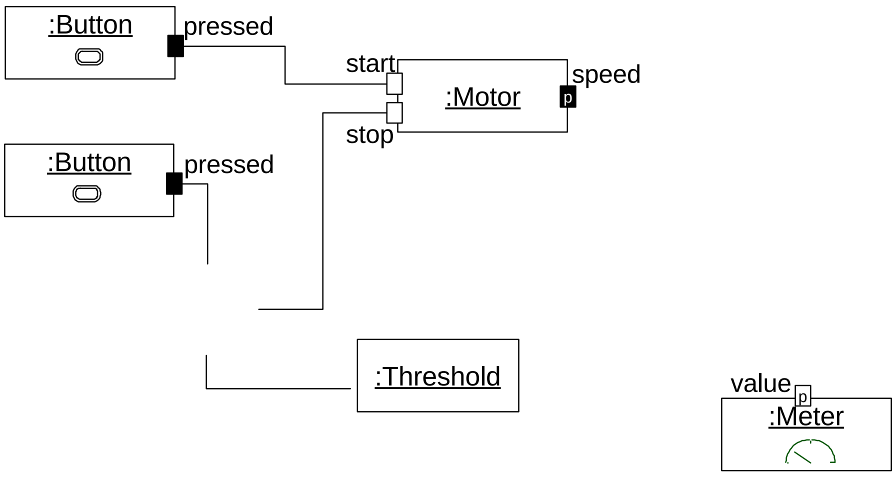


Figure 7.9 Component ports and event and property connectors.

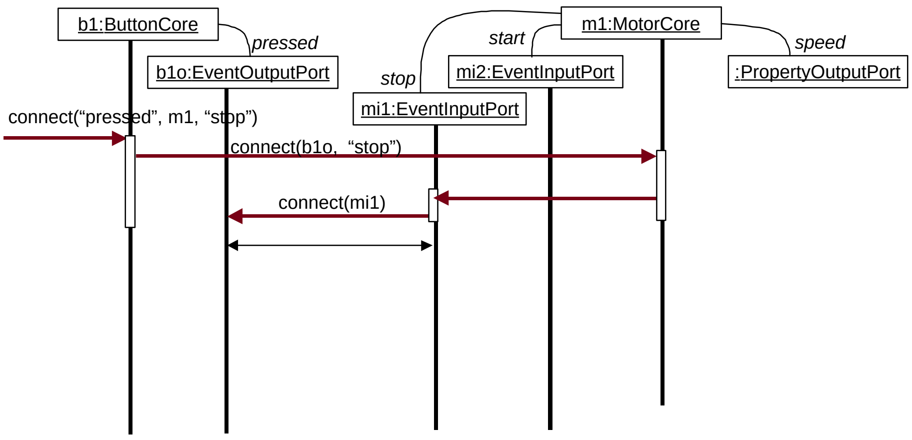
To illustrate an example of a reconfigurable system, Figure 7.9 shows the plugs that connect a simple electronic system together, using the real-time UML instance/port or **capsule** notation. Ports are linked by **connectors**. A connector is not necessarily implemented as a chunk of software in its own right: it is often just a protocol agreed between port designers. We try to minimize the number of connector types in a kit, so as to maximize the chances of any pair of ports being connected. In our example, two types of connector are visible, which we can call **event connectors** and **property connectors** – which transmit, respectively, plain events and observed attributes. You can think of the components as physical machines or the software that represents them *a piacere*. The button pressed interface always exports the same voltage (or signal) when it is pressed. The start and stop interfaces interpret this signal differently according to the motor's state machine. The point about this component kit is that careful design of the interface protocols and plug points allows it to be used for a completely (well not quite completely!) different purpose as shown in Figure 7.10. The members of this kit of parts can be rewired to make many different end products, rather like a construction toy. The secret of this reconfigurability is that each component incorporates its own adapter, which translates to a common 'language' understood by many of the other components. Such built-in adapters are called **ports**; they are represented by the small boxes in the figure.



by bolder lines than event connectors.

Once we have understood what the ports represent, we can get on and design useful products from kits of components, without bothering about all the details of registering interest, notification messages, and so on.

Ports can contain quite complex protocols. Each component must be well-specified enough to be usable without having to look inside it. So let's consider one way a port might work, as shown in Figure 7.11. An output port provides messages that allow another object to register interest in the event it carries; when the relevant event occurs (for example, when the user touches a button component) the port sends a standard 'notify' message to all registered parties. An input port implements the Listener interface for these notify messages; when it receives one, it sends a suitable message into the body of its component. For example, the 'start' port of the Motor component, when it receives a standard notify( ) message, will pass start ( ) to the principal object representing the motor.



course, these definitions have to be standardized before the components themselves are written. Common object types (int, Aeroplane, etc.) should be understood by all kit members. The kit architecture is the base upon which component libraries are built. Applications may then be assembled from the components in the library – but without the kit architecture, the whole edifice of CBD collapses. There are usually many architectures that will work for any scheme of connectors: the Java Beans specification provides a slightly different (and better performing) set of protocols to the ones we have described here.

Designing the architecture involves design decisions. A port should be realized as an object in its own right. The port connectors abstract away from the details of the port's protocol, but this must be specified eventually. For example, the sequence diagram in Figure 7.11 shows just one way in which the button-motor interface can be implemented. The property coupling port could be similarly implemented, but with regular update of new values.

Although we have illustrated the principle of the connector with small components, the same idea applies to large ones, in which the connector protocols are more complex and carry complex transactions. The current concern with Enterprise Application Integration can be seen as the effort to replace a multiplicity of point-to-point protocols with a smaller number of uniform connectors.

When specifying ports, or capsules, specify the types of parameters passed, the interaction protocol and the language in which the protocol is expressed. The interaction protocol could be any of:

- Ada rendezvous;
- asynchronous call;
- broadcast;
- buffered message;
- call handover;
- complex transaction;
- continuous dataflow;
- FTP;
- function call;
- HTTP, and so on.

The interaction language could be:

- ASCII, etc.;
- CORBA message or event;
- DLL/COM call;
- HTML;
- Java RMI;
- Java serialized;
- plain procedure call;
- RS232;
- TCP/IP;
- Unix pipe;
- XML;
- zipped.

7.3 Designing components

Writing in *Byte* John Udell tells us that 'Objects are dead!' and will be replaced by components. But he wrote this in May 1994, whereas objects didn't really hit the mainstream of information technology until around 1996/7. In fact, most CBD offerings that were around in 1994 didn't survive for very long after that: think of OpenDoc, OpenStep, Hyperdesk, etc. Probably the only significant survivor was the humble VBX. Currently, there are several understandings of what the term 'component' means. Some commentators just use it to mean any module. Others mean a deliverable object or framework: a unit of deployment. Still others mean a binary that can create instances (of multiple classes). Writers on object-oriented analysis tend to mean a set of interfaces with *offers* and *requires* constraints. The *requires* constraints are often called **outbound interfaces**. Daniels (2000) questions this and shows that while these aspects of a component must be defined, they do not form part of the component's contract. Szyperski (1998) defines a component as 'a binary unit of independent production, acquisition and deployment' and later 'A unit of composition with contractually specified interfaces and explicit context dependencies only'. We prefer: a unit of executable deployment that plays a part in a composition.

In many ways, VB is still the paradigm for component based development. Components were needed initially because OOPLs were restricted to one address space. Objects compiled by different compilers (even in the same language) couldn't communicate with each other. Thus arose distribution technologies such as RPCs, DCOM, CORBA, and so on. In every case interfaces were defined separately from implementation, making OOP only one option for the actual coding of the objects.

In the late 1990s ERP vendors that surfed on the crest of the year 2000 and Euro conversion issues did well because their customers needed quick, all-embracing solutions. When that period ended their market was threatened by a return to more flexible systems that were better tailored to companies' requirements. One vendor was even quoted as saying that a large customer should change its processes to fit the package because the way they worked was not 'industry standard'. This arrogance would no longer be tolerable after 2000 and the vendors were seen to rush headlong to 'componentize' their offerings; i.e. introduce greater customizability into them.

A lot of the talk about components being different from (better than) objects was based on a flawed idea of what a business object was in the first place. Many developers assumed that the concept of an object was coextensive with the semantics of a C++ class or instance. Others based their understanding on the semantics of Smalltalk objects or Eiffel classes or instances. Even those who used UML classes or instances failed to supply enough semantic richness to the concept: typically ignoring the presence of rules and invariants and not thinking of packages as wrappers (see Chapter 6). Further more, an object will only work if all its required services (servers) are present. This was the standpoint of methods such as SOMA from as early as 1993. SOMA (Graham, 1995) always allowed message-server pairs as part of class specifications. These are equivalent to what Microsoft COM terminology calls outbound interfaces.

From one point of view, outbound interfaces violates encapsulation, because the component depends on collaborators that may change and so affect it. For example, the client of an order management service should not have to know that this component depends on a product management one; an alternative implementation might bundle everything up into one object. For this reason John Daniels (2000) argues that collaborations do not form part of an object's contract in the normal sense. He distinguishes usage contracts from implementation contracts. The latter do include dependencies, as they must to be of use to the application assembler. That more research is needed in this area is revealed by another of Daniels own examples. Consider a financial trading system that collaborates with a real-time price feed. The user of the system should not care, on the above argument, whether the feed is provided by a well-known and reputable firm, such as Reuters, or by Price, Floggett & Runne Inc. But of course the user cares deeply about the reliability of the information and would differentiate sharply between these two suppliers. This example suggests that we should always capture the **effects** of collaborations as invariants or rulesets in the usage contracts, if not the collaborators themselves. The rules in this case would amount to a statement about the reputability of the information provider.

Current component based development offerings include CORBA/OMA, Java Beans/EJB, COM/DCOM/ActiveX/COM+, TI Composer and the more academically based Component Pascal/Oberon. What these approaches all have in common is an emphasis on composition/forwarding, compound documents, transfer protocols (e.g. JAR files), event connectors (single or multi-cast), metadata and some persistence mechanism. Differentiators between the approaches include their approaches to interface versioning, memory management, adherence to standards (binary/binding/etc.), language dependence and platform support.

From a suppliers point of view components are usually larger than classes and may be implemented in multiple languages. They can include their own metadata and be assembled without programming (!). They need to specify what they require to run. These statements could almost be a specification for COM+ or CORBA. Such component systems are not invulnerable to criticism. The size of a component is often inversely proportional to its match to any given requirement. Also, components may have to be tested late; an extreme case being things that can't be tested until the users downloads them (although applets are not really components in the sense we mean here). There is a tension between architectural standards and requirements, which can limit the options for business process change. Finally there is the problem of shared understanding between developers and users that we will discuss in Chapter 8. Szyperski discusses at length the different decision criteria used by each of infrastructure vendors and component vendors. He says nothing about the consequences for users. Can we deduce that users don't care about how components are developed? They certainly care about how they are assembled into applications.

Object modelling is about not separating processes from data. It is about encapsulation: separating interfaces from implementation. It is about polymorphism: inheritance and pluggability. It is about design by contract: constraints and rules. OO principles must be consistently applied to object-oriented programming, object-oriented analysis, business process modelling, distributed object design and components. However, there is a difference between conceptual models and programming models. In conceptual modelling both components and classes have identity. Therefore components are objects. Inheritance is as fundamental as encapsulation for the conceptual modeller. In programming models on the other hand components and classes do not have identity (class methods are handled by instances of factory classes). Thus components are not objects, class inheritance (or delegation) is dangerous and pluggability is the thing. So is there a rôle for inheritance? CBD plays down 'implementation inheritance'; but not interface inheritance but at the conceptual level this distinction makes no sense anyway.

When it comes to linking requirements models to analysis models, we can either 'dumb down' to a model that looks like the programming model (as most UML-based methods tend to do) or introduce a transformation process between the models (e.g. the SOMA to Catalysis transformation suggested in Chapter 8). The trade-off concerns the degree to which users and developers can share a common understanding.

7.3.1 Components for flexibility

Component based development is concerned with building extensible families of software products from new or existing kits of components. The latter may range in scale from individual classes to entire (wrapped) legacy systems or commercial packages. Doing this has hitherto proved an elusive goal for software developers. The trick is to realize that we need to define the interface protocols of objects in such a way that they can be plugged together in different ways. The number of interfaces needs to be small compared to the number of components. To improve flexibility these interfaces should permit negotiation in the same way as with facsimile machines or clipboard interfaces in Microsoft's OLE and the like.

An example may suffice to explain the point. Consider an hotel support system that is originally written for a small chain of Scottish hotels. The original is a great success and pays for itself quickly. But the price of success is rapid expansion and the company now acquires several more hotels in England, Wales, France and Germany. In Scotland rooms are always allocated to new arrivals on the basis of the empty room nearest the reception desk – to save visitors wearing out their shoes walking long distances. But the spendthrift and gloomy English want peace and quiet more than these savings; so that rooms are allocated alternately to leave empty rooms separating occupied ones when the hotel is not full. The Germans allocate rooms with French widows first. The different states involved also have different rules about wage payments. The system is amended piecemeal and soon there are several versions to maintain: one with nearest desk allocation and French payment rules, another with least-recently-used room allocation and UK staff payment laws and an *ad hoc* patch for management Xmas bonuses in Ireland and the Netherlands and so on. A maintenance disaster of some proportion!

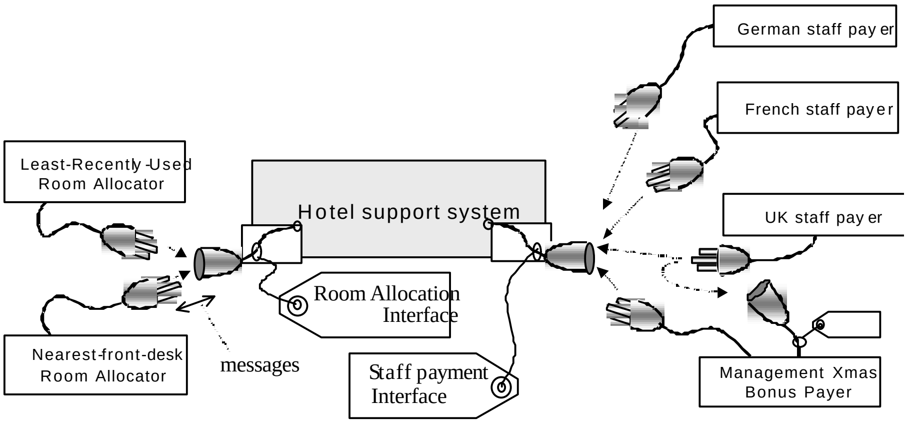

German staff payer
French staff payer
UK staff payer
Management Xmas Bonus Payer
Least-Recently-Used Room Allocator
Nearest-front-desk Room Allocator
Hotel support system
Room Allocation Interface
Staff payment Interface
messages

Figure 7.12 Plug points add flexibility.

A considerable improvement on arbitrary modification is shown in Figure 7.12. There is a basic framework, which does everything that is common between the requirements of all the hotels. Each separate variable requirement has been moved into a plug-in component: for example there are different room-allocators; and different staff payment components. This arrangement makes it much easier to maintain and manage variants of the system. We separate the rôles of the framework-designer and the designers of the plug-in components – who are not allowed to change the framework.

This makes it very clear that **the most suitable basis for choosing components is that they should correspond to variable requirements**. This is a key rule which people sometimes forget, while still claiming to be doing component based development.

One problem is that it is not always easy to foresee what requirements will be variable in the future. The best advice we can give here is as follows.

- Follow the principle of separation of concerns within the main framework, so that it is reasonably easy to refactor.
- Don't cater for generalizations that you don’t know you are going to need: the work will likely be wasted. Observe the eXtreme Programming maxim: ‘You ain't gonna need it!’.
- Where you do need to refactor the framework to introduce new plug-points, make one change at a time, and re-test after each change.

7.3.2 Large-scale connectors

In the previous section, we introduced the idea of connectors, using a Bean-scale example to illustrate the principle; the connectors transmitted simple events and property-values. But we can also use the same idea where the components are large applications running their own databases and interoperating over the internet. Recall that the big advantage of connectors over point-to-point interfaces was that we try to design a small number of protocols common to the whole network of components, so that they can easily be rearranged. In our small examples, that meant that we could pull components out of a bag and make many end-products; for large systems, it means that you can more easily rearrange the components as the business changes. This is a common problem being faced by many large and not-so-large companies.

For example, our Hotel system might have a web server in Amsterdam, a central Reservations system in Edinburgh, a credit card gateway in New Zealand, and local room allocation systems in each hotel world-wide. We would like to define a common connector, a common ‘language’ in which they all talk to one another, so that future reconfigurations need not involve writing many new adapters. Typical events in our hotels connector protocol will be customers arriving and leaving, paying bills; properties will include availability of rooms.

The component kit architecture for such a network will have to specify:

- low-level technology – whether it will use COM or CORBA, TCP/IP, etc.;
- a basic model of the business – what types of object are communicated between the components, customers, reservations, bills, etc.
- the syntax of the language in which the model will be transmitted – XML is often the solution;
- business transactions – e.g. how a reservation is agreed between the reservations system and a local hotel;
- business rules – is a customer allowed to have rooms in different hotels at the same time?

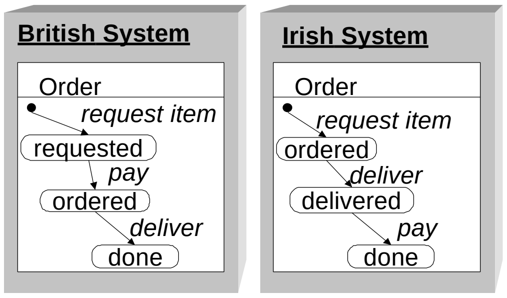


Figure 7.13 Incompatible business processes

This point about business rules is sometimes forgotten at the modelling stage. But it is very important: if one component of the system thinks a customer can have two rooms whereas another thinks each customer just has one, there will be confusion when they try to interoperate. And it is not just a question of static invariants: the sequences in which things should happen matters too. For example, imagine a company that comprises two divisions in different states as a result of a merger. In Great Britain the business demands payment before delivery is made, whilst in Eire payment is demanded after delivery is made. The different business régimes can be illustrated by the two state transition models in Figure 7.13. Problems will arise if these systems pass orders to each other to fulfil, because when a British customer orders a widget from the Dublin branch, they pass the request to the British system in the ordered state. That system assumes payment has been made and delivers – so that lucky John Bull never pays a penny. Obversely, poor Paddy Riley, who orders from the London branch is asked to pay twice.

7.3.3 Mapping the business model to the implementation

Catalysis provides specific techniques for component design. Actions are refined into collaborations and collaborations into ports. Retrievals (see Chapter 6) are used to reconcile components with the specification model. Consider the situation illustrated in Figure 7.14. Here there are two course grained components for accounting and dispatch, but they are based on different business models. Any system that integrates them must be based on a type model that resolves the name conflicts in the base components. For example our model must define customer in such a way that the subsidiary customer and payer concepts are represented. We must also include invariants to synchronize operations.

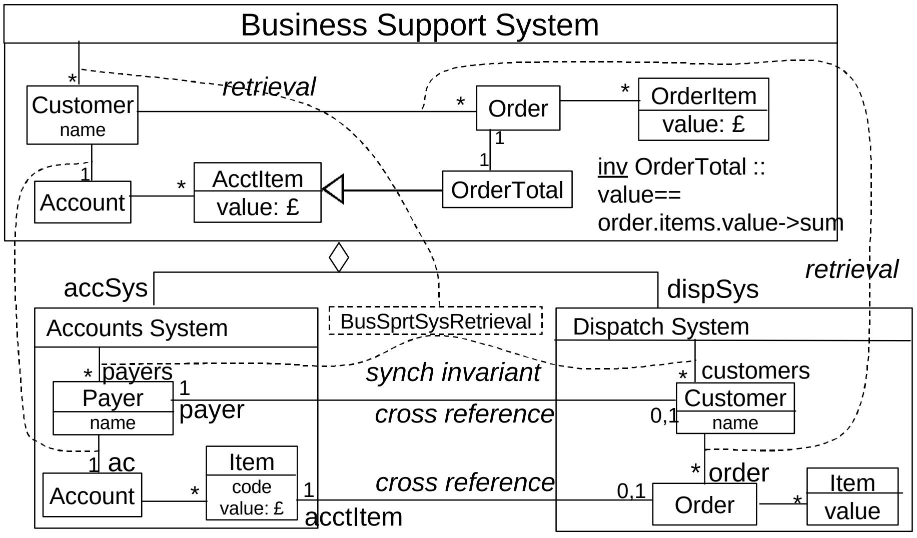


Figure 7.14 Retrieving a model from components.

Interfaces cannot be fully described in programming languages; the context in which they are used (pragmatics) is relevant. Often the only reason people accept lists of operations as specifications is because their names suggest the expected behaviour. We must say what components do as well as what they are; i.e. include type models, invariants, rulesets and protocols. The first three were covered in Chapter 6, now let us now see how we can describe the protocols rigorously.

Just as connectors must be encapsulated in objects, they can also be specialized and generalized. Templates, or frameworks, (see Chapter 6) are used to define connector classes. We define and specify each class and its interfaces. Next the connector classes are defined. Then message protocols for each class of connector can be defined using sequence and/or state diagrams. We must always ensure that a business model, common to all components, is clearly defined using a type model rich enough to define all parameters of the connector protocols. Points of potential variation should be handled using plug points. Business rules should be encapsulated by the interfaces.

Components will be used in contexts unknown to their designer(s) and so need far better 'packaging' than simple, fine-grain classes in class libraries. They should be stored and delivered with full documentation, covering the full specification and all the interfaces and ports and their protocols. It is mandatory that components should be supplied therefore with their test harnesses. Good component architectures require that components should be able to answer queries about their connexions at run time: complaining rather than collapsing under abuse by other software.

7.3.4 Business components and libraries

There seem to be three competing conceptions of a business object in use. Perhaps the oldest is due to Oliver Sims (1994) who meant something representable by an icon that means something sensible to users. Ian Graham's conception of a business object developed independently at around the same time. He meant any *specification object* that is mentioned in business process and task descriptions. The third conception emerges from discussion within the OMG. Their Business Object Modelling interest group promulgated the idea of a business object as a commonly used **software** component that occurs in the domain terminology. We have problems with this last definition insofar as it restricts business objects to the territory of commercial software vendors and excludes our ambitions of being able to model businesses, tasks and agents and of keeping our reusable objects totally machine and language independent. The OMG conception of a business object in its current form excludes intelligent agents from being business objects because IDL objects cannot easily express semantics equivalent to rulesets. We expect, however, that this will change at some point in the future.

Components occur at all scales from GUI frameworks – like the Java AWT, VBXs and OCXs – through program parts – such as Java Beans – up to entire applications crossing the language barrier delivered as Unix pipes or COM and OLE components – such as Excel. As we leave the boundary of a single computer and enter distributed space-time we encounter components packaged by CORBA, DCOM and Java RMT. There are several companies now claiming to offer libraries of business objects in vertical markets. These mostly consist of code libraries, an early, well-known one being the Infinity model for financial instruments. We have already mentioned the componentized ERP packages that are emerging at the time of writing these words. IBM's San Francisco was an early attempt to provide a whole library of business level components and design patterns (Monday *et al.*, 2000; Carey *et al.*, 2000). The Infinity offering uses a proprietary, but relational, data model known as Montage as the sub-structure upon which its C++ object model is built. This leads to performance problems, especially when complex derivative instruments are involved; because the database has to do many small joins to compose these complex composite objects. There is also a degree of incompleteness, with several instruments not being included in the model; users are expected to complete the model themselves. However, the product has been successfully used by several banks and is well suited to the smaller such institutions or to those not specializing in high volume derivatives markets. Infinity do not sell their abstract object model; they only offer the code. Some other offerings, that do represent themselves as abstract object models of financial instruments are usually, in our experience, little more than re-badged data models.

Continuing with Finance as a typical domain for business object modelling, let us pause to consider what types of thing are suitable for modelling as business objects or not. The class of financial instruments seems to be a good candidate because every instrument has a clear, stable and unambiguous definition. However, there are well over 300 classes in even the crudest model of financial instruments. This means that building a stable classification structure is extremely difficult. Furthermore, many financial products are composites. This leads to two completely different ways in which instruments can be modelled. The simplistic way is to build a broad, deep classification structure, noting the composition structures of each instrument as it arises. We have built such models and found them useful, if unwieldy. This is the approach taken by Infinity. A possibly more elegant approach exploits a theorem of financial engineering that says that every instrument is a composite of fundamental commodities and options on them. This means that instead of relying solely on classification structure we can also exploit composition. At least one important and successful risk management system at the Chase Manhattan Bank uses this approach. However, these general classification problems are still very tricky and should not be tackled by the inexperienced modeller. Smaller classification structures, however, are good candidates for inclusion in your first library of reusable specifications.

Corporate actions are a very good example of ideal candidates for business objects. There are (arguably) exactly 42[8] of them and they can easily be classified.

[8] Possibly answering a question posed by Douglas Adams.

Figure 7.15 shows a fragment of this classification structure. It can be seen immediately that even this structure is reasonably complex. Imagine what a structure with 350 classes would look like!

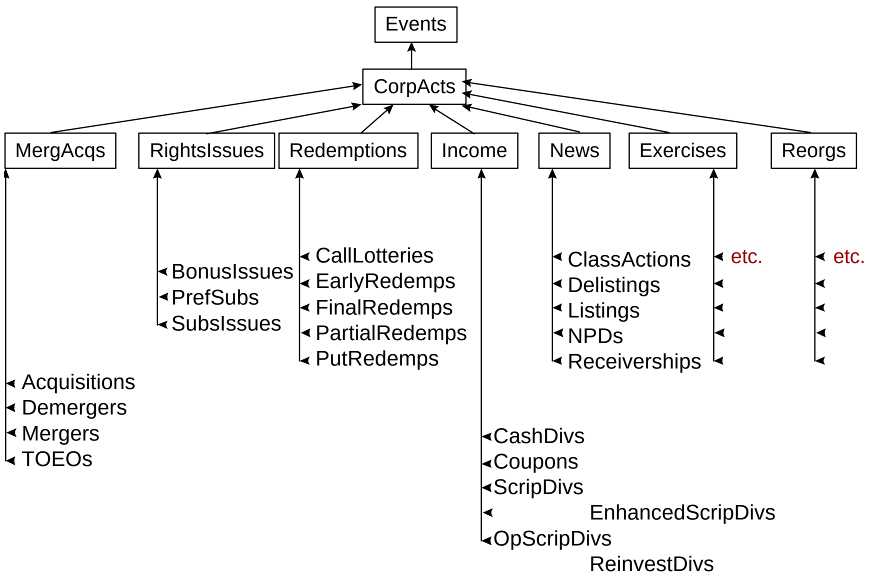


Figure 7.15 Classifying corporate actions.

On the face of it, pricing algorithms appear not to be good candidates because they are pure function and, of course proper objects must have data too. Reasoning in this way, the product should know how to price itself. However, in Finance there are many, many ways to price the same product, all of the algorithms being subject to evolution as new discoveries are made in financial engineering. It is therefore crazy to pack all these algorithms into Products. Here is a definite case where algorithms should be objects; these of course are nothing more nor less than function libraries resurrected in object-oriented form. And why ever not!

Events, such as trades, payments, cash flows, etc., are more problematical. Trades go through complex life-cycles that could perhaps be represented by state machines. We are thinking of states such as: quoted, agreed, affirmed, confirmed, matched, settled, etc. The difficulty here is that different instruments have very different life-cycles: foreign exchange settles quite differently from equities. The question is whether we should attempt to design a business object Trades that covers all eventualities or have different ones for the different kinds of instrument. Our instinct as object designers tells us that a single Trades class should refer to a complex instrument hierarchy with each instrument responsible for knowing its peculiar life cycle. However, we have already mentioned that the instrument hierarchy is extremely complex, so that making it more so may be considered unattractive. In practice, this is a difficult decision that will be taken according to the circumstances and judgment of different organizations.

But what of our customers, regulators, and so on? These are the worst candidates for business objects that we can think of because there is no such thing as a customer. Look at your own company's computer systems. The likelihood is that every system has a different definition of what a customer is. This is not because all past generations of developer were brain dead. It is because each department relates **differently** to its customers. Looked at one way, customerhood is a *rôle* adopted by a legal entity. Equivalently we can say that customerdom is a *relation* between your organization and another. Thus your business object library should have what Martin Fowler would call a Parties class and a series of relations representing concepts like account customers, overseas customers, etc.

Business processes are often advanced as candidates for modelling as business objects. This is, we feel, both right and wrong. Having actual software objects to represent a process is only apposite for the simplest processes and can lead to horribly rigid workflow systems in our experience. However, the approach presented in the next chapter does use objects (agents, conversations and actions) to model business processes, albeit indirectly. This gives more flexibility and expressive power. Beware therefore of people selling 'business process objects'.

There seem to be several current conceptions of what a business object model is. We regard one as a model of the domain concepts expressed in terms of objects. Of course, it is not (in this sense) a model of the business and its processes. Neither is it a pure system model (until refined by logical design). Some commercial offerings labelled 'vertical business object model' are little more than re-badged data/process models, replete with false assumptions about business processes taken from the small number of other organizations that were used to finance the original development. Creating a business object model of your own may be thus a better solution. It will provide a tailor-made resource and gives you more control. Such a strategy permits you to optimize performance if necessary, for example by choosing appropriate database technology. Your approach can take advantage of new technology as these innovations come along. However, it can be very expensive.

Business objects have been hyped. If you are considering buying a commercial business object model, then ask yourself if the model fits your business or if you will have to adapt to its foibles. Beware particularly of data models in disguise as object models. These are common. Beware too of models extracted from coded implementations, especially when the implementation is in some non-object-oriented language such as PowerBuilder or Visual Basic. Ask the supplier to give you the history of the product with this consideration in mind. Ask too, if the model is implemented, how the database is implemented and designed: is performance going to be adequate for the type and scale of application you have? Use binaries only when you trust them. Buy or build the right middleware; and be prepared to pay for it. An architecture based on sound middleware decisions is the only basis for profitable gains from re-use.

The emphasis of CBD is on deployment architecture and packaging, and new design methods such as Catalysis are clearly indicated; but Component Based Analysis is something of an oxymoron. What is needed is a mapping from sound BPM/OOA techniques to Component Based Design which derives use cases and actions from process models. Catalysis 'refinement' techniques then ensure traceability down on through the modelling to implementation.

It is still unclear whether there really is a mature component market. We have OCXs, we have Java Beans, we have San Francisco even; but undoubtedly the range and types of component on offer will grow. As this happens we will increasingly distinguish component building from application assembly, both requiring new development process models. We will also need to make methodological changes.

In our new methods inheritance must treated differently at the conceptual and component levels. It remains a fundamental concept for conceptual modelling and knowledge representation, and is readily understood by users. However, inheritance is not the ideal way to build flexibility into components as we have seen; plug points based on the patterns we have presented give far greater flexibility. We are, of course, endangering seamlessness slightly, but the price is well worth paying. In the same way the principle of substitutability is crucial for the design of flexible components, but it is often counter-intuitive during conceptual modelling.

CBD is often contrasted with object-oriented development, as though they were quite different. One reason for this is that some (bad) object-oriented programming practice concentrated on very small objects at the expenses of enterprise level ones and completely ignored collaborations (outbound interfaces) and interface semantics (rules, etc.). Emphasis was on weak object models based on languages rather than the sort of semantically rich conceptual models that OOA methods like SOMA always advocated. In this sense CBD merely tells us where some of us got OO wrong.

We may also conclude that successful CBD will require a heightened focus on architecture and patterns together with the semantically rich object modelling techniques mandated by methods such as Catalysis and SOMA. An organization adopting CBD should always discuss architecture and reuse up front. It must also **manage** the reuse process, as we discuss in Chapter 9. The main methodological changes are at the level of design, architecture and implementation.

Remember that models are powerful representations of knowledge, not just computer stuff! Object models can be used to represent many things: any kind of object knowledge that is. The semantics of the modelling language you use must therefore be powerful enough to express this knowledge; which means that it should include class invariants (ideally rulesets), usage links and other features necessary to model business processes. This is because it is never enough to just model the computer system. A good model should include users' conversations, contracts, goals and tasks. Take control over the specification yourself rather than relying entirely on external suppliers.

Remember also that greenfield sites are the exception; most components will be used within a context of existing systems and these must be modelled as design level components. In many cases there will be a strong need for a repository of existing and buyable components and previous specifications (processes, objects, etc.). There should be a relatively fixed architectural framework

7.4 Summary

This chapter surveyed the emerging fields of software architecture, patterns and component design. We argued that architecture was more than just structure and must include vision or rationale. We also argued for its importance.

Several design, analysis, architectural and organizational patterns were introduced and we touched on the idea of pattern languages and related concepts like problem frames.

Building on this base, we made extensive use of patterns and concepts from Catalysis, such as retrievals, to show how to design robust, flexible component based systems. Out emphasis was on techniques for decoupling components. Finally we looked at some issues surrounding the emergence of a market for components.

7.5 Bibliographical notes

Shaw and Garlan (1996), Bass *et al*. (1998) and Bosch (2000) all discuss software architecture from a broadly structural perspective. Alan O'Callaghan's series of articles in *Application Development Advisor* [sic] provide a different view closer to the one expressed in this chapter.

Design patterns were introduced by Gamma *et al*. (1995). Buschmann *et al*. (1996) cover these together with architectural patterns. Pree (1995) is another early contribution in this area and introduced the idea of metapatterns. Fowler (1996) discusses analysis patterns; the idea of which was earlier suggested by Coad (1992). The annual proceedings of the PLoP conferences (Coplien, and Schmidt, 1995; Vlissides *et al*., 1996) are full of interesting exegesis and new patterns including the earliest work on organizational patterns (Coplien, 1995). Beck (1997) is an excellent exposition of how to use patterns in the context of Smalltalk, well worth reading by those not interested in that language. Coplien (1992) describes C++ idioms. His new book …..?? … Richard Gabriel …. Mowbray and Malveau (1997) discuss patterns for CORBA implementations. Much current work on patterns has been inspired by the work of Christopher Alexander and his colleagues (1964, 1977, 1979) in the built environment. Lea (1994) gives a concise summary of Alexander's ideas.

Madsen, et al. (1993) discuss a different conception of a pattern in the context of the BETA 'pattern oriented' language. This makes patterns a generalization of classes but it is still unclear what this conception has to do with the patterns we have described in this chapter.

Gardner *et al*. (1998) discuss their idea of cognitive patterns based on the task templates of the Common KADS knowledge engineering method. The idea is that common problem solving strategies, such as diagnosis, planning and product selection, can be classified as patterns that show how each kind of inference proceeds.

Brown *et al*. (1998) discuss **anti-patterns**: common solutions to frequently occurring problems that **don't** work – and provide suggested solutions. The idea has much in common with the 'software ailments' of Capers Jones (1994).

OORAM (Reenskaug *et al*., 1996) is a method, tool and language in which collaborations and their composition are the central design mechanisms, it is discussed briefly in Appendix B.

D'Souza and Wills (1999) describe the Catalysis method for component based development. Wills (2001) is a more recent and (we hope) readable introduction. Cheesman and Daniels (2000) describe a simple process for specifying software components and component-based systems, using UML notations and drawing on ideas from Catalysis. Substantial parts of Sections 7.2 and 7.3 are based on a TriReme technical white paper written by Alan Wills.

7.6 Exercises

TBD.

Compare two architectural styles from the following list: layers, blackboards, pipes and filters, CORBA, peer-to-peer, client-server

Write a pattern showing how to

decoupling